Automated Code Annotation with LLMs for Establishing TEE Boundaries

Varun Gadey*, Melanie Götz*, Christoph Sendner*, Sampo Sovio†, and Alexandra Dmitrienko*
*University of Würzburg, Germany
†Huawei Technologies

***Abstract*—Modern systems increasingly rely on Trusted Execution Environments (TEEs), such as Intel SGX and ARM TrustZone, to securely isolate sensitive code and reduce the Trusted Computing Base (TCB). However, identifying the precise regions of code especially those involving cryptographic logic that should reside within a TEE remains challenging, as it requires deep manual inspection and is not supported by automated tools yet. To solve this open problem, we propose LLM based Code Annotation Logic (LLM-CAL), a tool that automates the identification of security-sensitive code regions with a focus on cryptographic implementations by leveraging most recent and advanced Large Language Models (LLMs). Our approach leverages foundational LLMs (Gemma-2B, CodeGemma-2B, and LLaMA-7B), which we fine-tuned using a newly collected and manually labeled dataset of over 4,000 C source files. We encode local context features, global semantic information, and structural metadata into compact input sequences that guide the model in capturing subtle patterns of security sensitivity in code. The fine-tuning process is based on quantized LoRA—a parameter-efficient technique that introduces lightweight, trainable adapters into the LLM architecture. To support practical deployment, we developed a scalable pipeline for data preprocessing and inference. LLM-CAL achieves an F1 score of 98.40% and a recall of 97.50% in identifying sensitive and non-sensitive code. It represents the first effort to automate the annotation of cryptographic security-sensitive code for TEE-enabled platforms, aiming to minimize the Trusted Computing Base (TCB) and optimize TEE usage to enhance overall system security.**

I. Introduction

In today's interconnected software landscape, protecting sensitive operations within applications has become a critical challenge. Although modern computing platforms come equipped with advanced security features like isolation, access control, and memory protection, these measures fundamentally depend on the integrity of the underlying software layer—typically an operating system or hypervisor. This underlying software forms the core of the Trusted Computing Base (TCB), the set of components that must remain secure for the system to function safely. Consequently, any compromise of this software layer endangers the security of all applications and data within the system.

Network and Distributed System Security (NDSS) Symposium 2026
23–27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230709
www.ndss-symposium.org

However, operating systems and hypervisors are vulnerable to a variety of attack vectors, including rootkits [1], kernel exploits [2], [3], hypervisor vulnerabilities [4], [5], and malicious alterations during installation [6], [7]. Such compromises allow attackers to bypass platform security measures altogether, enabling them to manipulate memory [8], extract sensitive data [9], alter control flows [10], disable security checks [11], and inject malicious code into applications at runtime [12]. This issue is exacerbated by the inherent complexity and size of modern system software, which make it nearly impossible to formally verify or exhaustively test against all potential vulnerabilities.

A promising approach has emerged to address the growing risk of system software compromise: Dividing applications into security-sensitive and non-sensitive components and executing only the sensitive components under the protection of Trusted Execution Environments (TEEs). TEEs are enabled by hardware-security extensions that provide hardware-based isolation for encapsulating sensitive code and data. Two prominent examples of TEEs are Intel SGX [13] and ARM TrustZone [14], each employing distinct models suited to their hardware architectures.

Intel SGX enables the creation of isolated enclaves, designated sections of an application that execute sensitive code independently of the main system. This protects enclave-resident code and data even from the operating system and hypervisor, ensuring that these components remain secure against privileged software attacks. Although SGX can technically support entire applications or even operating systems within an enclave, this increases the TCB, raising the risk of exploitable vulnerabilities. From a security perspective, keeping the TCB within an enclave as small as possible is generally preferable, including only the most critical operations.

ARM TrustZone divides the system into two worlds: Secure and normal. Security-critical operations run in the secure world, isolated from non-sensitive processes in the normal world. Switching between worlds is managed via the Secure Monitor Call (SMC), which invokes the Secure Monitor [15]. The Secure Monitor ensures secure state transitions, protecting memory and registers from data leakage. Resources in the secure world are limited, with memory varying from a few to tens of megabytes, depending on device configuration and hardware capabilities [16], [14].

For applications that leverage TEEs, the TCB is confined to the code within the enclave (as in Intel SGX) or to the Secure Monitor and code operating in the isolated secure world (as in ARM TrustZone). By narrowing the TCB to these isolated components, TEEs provide stronger security assurances for critical operations, reducing the risk of vulnerabilities in system software and creating a more robust foundation for protecting sensitive data.

However, integrating a TEE into an existing application is complex and requires specialized security expertise. Determining which components are genuinely security-sensitive and which can remain in the regular execution environment involves a labor-intensive analysis of the entire code base. Even the definition of "security-sensitive code" can be challenging, as it often depends on the specific notion of the application and the threats it faces. In this work, we focus on a well-defined subset, code that implements cryptographic functionality and the surrounding logic that processes sensitive inputs and outputs requiring isolation. As a result, automating this identification process remains an open challenge, and no fully automated tools currently exist to assist developers in this task. This highlights the critical need for developer teams to have a deep understanding of their applications and the associated security risks. The situation underscores the urgent demand for reliable identification tools that empower developers to utilize TEEs effectively while maintaining robust security. Such tools could alleviate the burden of manual analysis, minimize the risk of oversights, and ultimately enhance the overall security posture of applications leveraging TEEs.

Given the lack of tools for automated code annotation, many developers move entire applications into the secure environment, aided by tools like Graphene [17], Haven [18], and SCONE [19]. Graphene and Haven enable legacy applications to run in secure enclaves, with Haven focusing on cloud environments. SCONE secures containerized applications, specializing in microservices. These tools automate the migration of *full* applications into TEEs, which leads to introducing unnecessary code into TCB, leading to the blown up TCB and the increased risk of vulnerabilities. While a few approaches such as SOAAP [20], DATASHIELD [21], and Yin Lin et al [22] have explored the possibility of automatically splitting the code in the context of TEE integration [21], [22] and enforcing the principle of least privilege [20], they focus on automatically partitioning the manually annotated code rather than facilitating annotations.

In this work, we address this critical gap by introducing the first tool for the automated annotation of security-sensitive code, with a specific focus on cryptographic implementations. We present LLM-CAL — an LLM-based Code Annotation Logic system capable of analyzing C programs to automatically identify and annotate code regions that require TEE protection. As the first solution of its kind, LLM-CAL is designed to examine an entire codebase and determine which code lines (mapped into functions) are security-sensitive and should be relocated to a TEE, and which can safely remain in the regular execution environment. By automating this process, LLM-CAL enables developers to optimize their applications for TEE deployment, reduce the Trusted Computing Base (TCB), and enhance overall system security, without the need for time-consuming and error-prone manual code inspection.

Contributions: In more detail, we make the following contributions:

- **Cryptex code notion and dataset.** We define security-sensitive code in terms of cryptographic function usage and the confidentiality and integrity of its inputs. We refer to it as cryptex code[1] in the rest of the paper. Cryptex code refers to all program logic that directly interacts with cryptographic primitives, as well as the broader set of code that handles sensitive inputs to or outputs from these primitives. This includes pre-processing, parameter handling, and post-cryptographic data flows. Based on this definition, we constructed a comprehensive dataset of 4010 open-source C files involving cryptographic operations. This annotation effort, requiring significant domain expertise and over five person-months of manual analysis, addresses a key gap in automated TEE integration research. To support reproducibility and future research, we commit to releasing this dataset publicly (https://github.com/VarunGadey/llmcal_dataset).
- **Data preparation and LLM Fine-tuning** We present LLM-CAL, a scalable and memory-efficient fine-tuning framework that leverages quantized LLMs, specifically the Gemma-2B backbone with QLoRA-based [24] low-rank adapters. Our input construction framework integrates local context of each code line, global function-level semantics, and structured metadata to make robust and generalizable secure line annotations.
- **Evaluation and case studies.** We evaluated LLM-CAL using Gemma-2B [25], Llama-7B [26], Codegemma-2B [27] models at fine-grained line level accuracy. LLM-CAL achieves 99.04% accuracy, 97.50% recall, 99.40% precision, and an F1 score of 98.41% with best QLoRA configuration in identifying both cryptex code and non-cryptex code lines. At the function level, LLM-CAL achieves 100% identification rate with zero false negatives and positives. We further validate LLM-CAL through three real-world case studies: (i) a Bitcoin utility tool with complex cryptographic workflows, (ii) firmware code from TZ-DATASHIELD [21], and (iii) out-of-distribution cryptographic code from alternative libraries such as mbedTLS. Lastly, to demonstrate the robustness of LLM-CAL, we curate a dataset of alternative cryptographic code and show statistically significant results on out-of-distribution samples. In each case, LLM-CAL reliably highlights security-sensitive regions while suppressing benign lines, demonstrating strong generalization across deployment contexts. Altogether, our LLM-CAL achieved a high TCB reduction of 81.20%

[1]Cryptex – a combination of "crypto" and "context". The term was coined in Dan Brown's "The Da Vinci Code" [23].

on test set, with only 0.52% inflation of TCB size due to misclassification by the tool.

Our novel dataset and LLM-based model form a robust framework for the automated annotation of cryptex code (i.e., cryptographic security-sensitive code) for TEEs. By automating a previously labor-intensive manual task, LLM-CAL helps minimize the TCB by excluding non-essential code from trusted regions, thereby reducing the attack surface. With the LLM-CAL framework, developers can efficiently leverage TEEs for improved performance and security while avoiding the risks of overly broad code inclusion.

Outline. The remaining part of the paper is organized as follows. Section II introduces key background concepts. Section III defines the problem of cryptex codeidentification, highlighting the challenges involved. In Section IV, we present our LLM-CAL approach in detail. Section V presents the evaluation results, showcasing method's effectiveness through various metrics and case studies. Section VI reviews the related literature. Finally, Section VII concludes the paper.

II. Background

In this section, we provide essential background on Large Language Models (LLMs) and their fine-tuning.

A. Large Language Models

Large Language Models (LLMs) are deep neural networks based on the Transformer architecture, trained on extremely large-scale textual and source code corpora. The transformer architecture of LLMs relies heavily on the attention mechanism, which allows it to learn how to focus on parts of the input while ignoring the rest. Specifically, the transformer architecture uses self-attention, which operates within the layer rather than taking input from another layer. These self-attention mechanisms captures complex syntactic and semantic patterns, which leads to strong performance across various tasks including question answering, code summarization, and code understanding.

In recent years, several open-source LLMs such as LLaMA [26], Gemma [25], and CodeGemma [27] have emerged as strong foundation models for downstream language and code understanding tasks and heavily leveraged in research environments for their promising results. These models are pre-trained on a broad mixture of natural language and programming languages and can be further adapted to specialized domains through task-specific fine-tuning.

B. Fine-Tuning

As described in II-A, although LLMs are pre-trained as general-purpose models, they can be adapted to specific tasks through fine-tuning on labeled data. While LLMs are typically implemented as sequence-to-sequence models, their architecture can be extended for downstream tasks. For instance, in classification tasks, a classification head is added to the model, and during fine-tuning, the model learns to map contextual embeddings to discrete output labels.

Full Fine-Tuning: Full fine-tuning involves updating all parameters of a pre-trained model using task-specific labeled data. This approach allows the model to learn domain-specific patterns while leveraging its existing knowledge. However, for modern LLMs with billions of parameters, full fine-tuning presents several challenges. First, updating the entire model increases the risk of overfitting to the limited task-specific data, particularly when the labeled dataset is small and highly domain-specific. Second, full fine-tuning requires substantial computational resources, including high memory bandwidth and prolonged training time. These constraints often make full fine-tuning infeasible in practical scenarios.

Parameter-Efficient Fine-Tuning (PEFT) To address these limitations, *PEFT* techniques [28] are emerged, which update only a small subset of model parameters while keeping the rest of the model frozen. PEFT significantly reduces the computational burden and risk of overfitting, making it ideal for use cases with limited computational resources or niche application domains.

Low-Rank Adaptation (LoRA) One of the most widely used techniques in PEFT is LoRA [29]. LoRA reduces the number of trainable parameters by introducing a low-rank update to existing model weights instead of modifying the full weight matrices. Given a weight matrix $W \in \mathbb{R}^{d\times k}$ in a neural network layer, LoRA introduces two smaller trainable matrices:

$$A \in \mathbb{R}^{d\times r}, \quad B \in \mathbb{R}^{r\times k}, \quad \text{where } r \ll \min(d,k)$$

The updated weight matrix becomes:

$$W' = W + \Delta W = W + \alpha AB$$

Here, α is a scaling factor, and the original weight matrix W remains frozen during fine-tuning. Only A and B are trained, which drastically reduces the number of learnable parameters while still allowing the model to adapt effectively to the target task. Moreover, at initialization, $\mathbf{AB} \approx \mathbf{0}$, the original pre-trained behavior of the model is still preserved.

Quantized Low-Rank Adaptation (QLoRA) QLoRA [24] extends LoRA to support the fine-tuning of large language models stored in a 4-bit quantized format. This approach allows the fine-tuning of models with billions of parameters on consumer-grade hardware by significantly reducing memory usage. QLoRA uses a quantization function $\mathcal{Q}(\cdot)$ that maps full-precision weights to lower-bit representations:

$$\tilde{W} = \mathcal{Q}(W)$$

During fine-tuning, the quantized weights $\tilde{W}$ remain frozen, and LoRA-style low-rank updates are applied on top:

$$\tilde{W}' = \tilde{W} + \alpha AB$$

This combination of quantization and low-rank adaptation makes QLoRA particularly suitable for fine-tuning large models in resource-constrained environments where access to high-end hardware may be limited.

Dynamic Offset Rank Adaptation DORA [30] is another technique and a recent advancement in PEFT that improves

traditional low-rank adaptation by dynamically adjusting the rank of the A and B matrices during training. Instead of using a fixed-rank approximation like LoRA, DORA adaptively increases capacity only where needed, enabling more flexible and efficient fine-tuning with minimal overhead.

III. Problem Statement

TEE-enabled applications isolate cryptex code within TEEs, leaving non-critical code in the normal environment. However, identifying this boundary requires significant domain and security expertise. Our work aims to automate this process by developing a labeled dataset and training an LLM to recognize cryptex codepatterns in unseen code.

A central challenge in this effort is the absence of a clear, universal definition of cryptex code, as its meaning can vary across application contexts and threat models. This section establishes the foundation for addressing this issue: In Section III-A, we propose a definition of cryptex code that meets the needs of a wide range of applications. Additional challenges stem from leveraging LLMs for this task, which requires a deeper semantic understanding than traditional code analysis. We discuss them in Section III-B.

A. Notion of Cryptex Code

While the meaning of security-sensitive code can vary across different applications, we argue that it is reasonable to define it based on the use of **cryptographic operations** (i.e., cryptex code). Indeed, core security mechanisms, such as authentication, access control, digital rights management, etc., fundamentally rely on **cryptographic primitives** when implemented securely. For instance, when a program performs password-based user authentication, it would normally require hashing of provided inputs to compare them with stored password hashes, ensuring that sensitive data like passwords is never exposed in plain text. In another example, secure communication protocols, such as TLS/SSL, rely on cryptographic operations like key exchange and encryption to protect data in transit between parties. Similarly, digital signatures used in software distribution rely on public-key cryptography to verify the integrity and authenticity of software packages.

Moreover, data encryption and decryption operations within storage systems are inherently cryptex code, as they protect confidential information at rest. For example, when an application encrypts user data before storing it in a database, the encryption key and the method of encryption constitute cryptex code. In the context of digital rights management (DRM), cryptographic techniques such as public-key encryption and digital certificates are used to ensure that content, such as movies, music, or e-books, is only accessible to authorized users. A typical use case involves encrypting digital content with a unique key and binding that key to the user's device, so only the authorized device can decrypt and access the content, preventing unauthorized redistribution or piracy.

Thus, any code that handles or processes cryptographic keys, encrypts/decrypts data, or performs operations such as hashing and digital signing can be considered cryptex code. Hence, in software projects that integrate TEEs, cryptographic functions must always be executed within the TEE to ensure the trustworthiness of their outputs.

Beyond the cryptographic functions themselves, our notion of cryptex code extends to their input. Cryptographic operations act as anchors for identifying broader regions of code that process or influence sensitive information. Inputs and outputs to cryptographic functions, such as passwords, encryption keys, or data to be encrypted, are equally critical. If such inputs are exposed, modified, or accessed by untrusted code, the security guarantees of the cryptographic operation can be invalidated. Therefore, these inputs also require protection and should be handled within the TEE.

From a dataflow perspective, cryptographic functions act as *sinks* in the flow of sensitive information. Our definition of cryptex code, therefore, extends to the entire dataflow path that leads to these sinks. This includes all code that generates, processes, or transmits inputs to cryptographic operations. By securing every element along this path, we aim to provide comprehensive protection for any information flowing into cryptographic functions, whether it be security-sensitive data, passwords, or cryptographic keys. While our definition of cryptex code is grounded in the use of cryptographic functions as sinks, it ultimately encompasses the full data flow, which may involve components such as access control, authentication, personal data handling, digital rights management, and other security-critical logic. We contend that this broad definition is adaptable to a wide range of applications.

B. Challenges

In developing a reliable framework for identifying cryptex code, several challenges emerge that must be addressed to enhance the effectiveness of automated tools and methodologies. These challenges stem from the inherent complexities of software systems, the nuances of data flows, and the limitations of current analysis techniques. Below, we outline the primary challenges encountered in this endeavor, each highlighting critical aspects of the problem.

CH I - Complexity of the Cryptex CodeNotion: The first challenge lies in accurately identifying sources of cryptex data flows related to cryptographic operations. Each function must be evaluated to determine if its execution could impact the confidentiality and integrity of sensitive data used in cryptographic operations. Assessing whether each function could serve as a valid entry point for a security-sensitive data flow is crucial to ensure that placing it outside the TEE boundary does not compromise the security of the inputs involved. This requires analyzing all possible function invocations within the application to determine if any could violate security guarantees when isolated outside of TEE.

Static analysis faces limitations due to its lack of runtime context, meaning it cannot account for dynamic factors like user inputs or environmental variations, which makes modeling complex control flow and data dependencies challenging. Dynamic execution, on the other hand, struggles with

incomplete coverage, as it is practically impossible to exercise every possible code path, particularly in large applications. As a result, accurately identifying cryptex code remains a significant challenge.

CH II - Dataset: The second major challenge is the lack of existing datasets to support the training and validation of models for identifying cryptex code. Reliable identification tools require a substantial volume of labeled data, but no publicly available datasets categorize code by security sensitivity, particularly for cryptographic operations and TEE integration. Manually creating such datasets is labor-intensive and requires expert knowledge to accurately identify and label cryptex code elements, such as specific code sections, functions, or data flows that need protection to maintain security guarantees.

CH III - Long-Range Semantics in Limited Context Windows: Cryptex code often depends on information spread across multiple lines or distant function calls. This creates long-range semantic dependencies that are difficult to capture within the fixed-length input constraints of large language models. Unlike local vulnerability patterns, secure code identification requires broader contextual understanding, which is challenging to encode in a single input sequence.

In cryptex code, subtle semantics often emerge from adjacent lines such as preparatory checks, assignments, or control statements that precede or follow a target code line. When these local interactions are not fully captured, models risk misclassifying code line that appears benign in isolation. Designing representations that capture such fine-grained, line-to-line dependencies within a constrained input sequence presents a sharp modeling challenge.

CH IV - Scaling Large Models to Domain-Specific Tasks: Modern LLMs are designed for general-purpose understanding and typically require substantial compute and memory for both training and inference. Applying such models to narrow, domain-specific tasks like cryptex code identification poses challenges in terms of scalability and efficiency. Moreover, compressing models through quantization to reduce resource usage can lead to information loss, potentially diminishing their ability to generalize to fine-grained security contexts.

IV. LLM BASED CODE ANNOTATION LOGIC

In this section, we introduce LLM-based Code Annotation Logic (LLM-CAL), the first tool to address the critical gap of automating the annotation of cryptex code and facilitating the seamless integration of Trusted Execution Environments (TEEs) into existing applications. LLM-CAL leverages LLMs, such as Gemma [25], Llama [26], and CodeGemma [27], and is designed to analyze C programs and automatically identify and annotate code regions that require TEE protection. Specifically, LLM-CAL targets cryptographic code and its related logic that handles sensitive inputs and outputs (i.e., cryptex code), forming the core of cryptex regions in TEE-enabled applications.

LLM-CAL formulates the task of identifying cryptex codeas a supervised binary classification problem at the line level. Adopting a sequence classification approach, each line of code is treated as an independent prediction unit. To enable this, we fine-tune LLMs using QLoRA [24] on a curated dataset of C code, annotated according to our definition of cryptex code (cf. Section III-A).

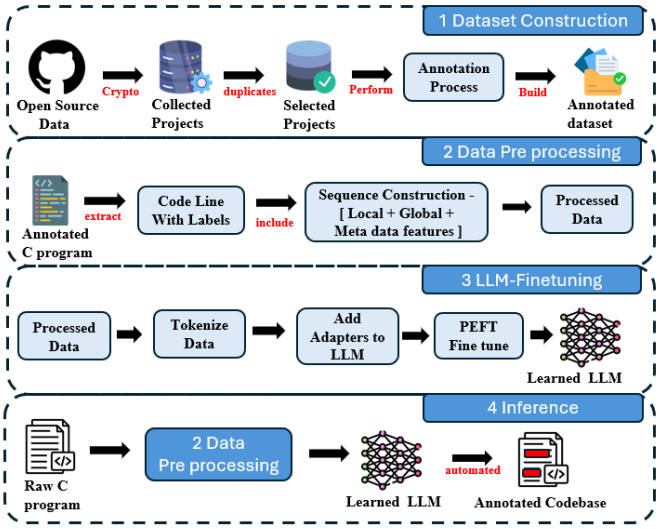

Fig. 1: LLM-CAL Overview

While LLM-CAL operates at the line level to capture fine-grained security-relevant details, TEEs are best leveraged at coarser granularities. Frequent context switches between trusted and untrusted environments can incur performance overhead, and attempting to isolate only small code fragments may increase complexity and inadvertently expose sensitive data through insecure paths. To address this, we aggregate the line-level predictions into function-level classifications, which define the boundaries for secure execution.

As the first tool of its kind, LLM-CAL can analyze an entire codebase to determine which code segments mapped to functions should be relocated to a TEE and which can safely remain in the normal execution environment. By automating this process, LLM-CAL empowers developers to streamline TEE deployment, reduce the Trusted Computing Base (TCB), and enhance system security without relying on time-consuming and error-prone manual inspection.

A. Threat Model

Our threat model targets adversaries typically assumed for TEE-enabled platforms. This implies a powerful adversary with kernel-level access to the system, who seeks to exploit software vulnerabilities in order to compromise the integrity or confidentiality of sensitive data and code. Using control over the system, the attacker aims to breach isolated components or extract sensitive information.

However, we assume that the adversaries cannot access or modify the hardware of the platform and its trusted computing base, i.e., the code running within the TEE. This is a typical assumption made for TEE-enabled systems. Side-channel attacks are also excluded, while they were shown to be effective

(e.g., [31], [32], [33], [34]), respective defense methods were developed (e.g., [35], [36], [37]) that can be deployed on a target system. Also, adversaries cannot interfere with the operation of LLM-CAL itself. This assumption is justified, as LLM-CAL is used during the code development phase prior to deployment, where the system is not yet exposed to adversarial influence.

B. LLM-CAL Design

The overview of the LLM-CAL's functionality is provided in Figure 1. Its design comprises four key phases: (i) Dataset construction; (ii) Input sequence construction; (iii) LLM fine-tuning; and (iv) Inference and code annotation.

- **Dataset Construction:** In order to fine-tune LLM-CAL, we curated a dataset of open-source C programs from GitHub [38], focusing on projects that make use of cryptographic libraries. We establish annotation guidelines and manually label code lines in hundreds of projects based on our definition of cryptex code, established in Section III-A, and address the challenge CH1 (Cryptex Code-Definition). This dataset forms the foundation for training and evaluating our models.
- **Input Sequence Construction:** Whether for fine-tuning or inference, LLM-CAL transforms raw C source code into structured input sequences compatible with LLMs. Each line of code is enriched with contextual information, including adjacent lines (local context), static analysis features derived from call graphs and data flow features extracted from code property graphs (global context), and metadata. These features help address challenges CH3 (Long-Range Semantics) and CH4 (Scaling Domain-Specific LLMs)
- **LLM Fine-Tuning:** We utilize QLoRA adapters for parameter-efficient fine-tuning of a pre-trained LLM. These adapters are integrated into the attention layers of the model architecture to update only a small subset of parameters. The model is then fine-tuned on our annotated dataset, learning to distinguish between cryptex and non-cryptex code patterns.
- **Inference and Code Annotation:** In the inference phase, we reuse the same input sequence construction pipeline to process previously unseen C code. The fine-tuned LLM evaluates each enriched line and outputs a confidence score indicating the likelihood of cryptex functions. Based on these scores, LLM-CAL generates annotated outputs highlighting code lines and corresponding functions that should be migrated to the TEE.

In the following, we describe the inner workings of every phase in more detail.

1) Phase 1 - Dataset Construction: As outlined in Section III-B, the successful implementation of a machine learning-based approach for modeling and automatically detecting cryptex code in previously unseen projects necessitates a comprehensive dataset wherein these functions and variables are explicitly labeled. To address CH2 (Dataset Constraints), we created such a dataset by collecting relevant open-source projects from GitHub [38] that utilize cryptographic libraries. We subsequently implemented a structured manual labeling process designed to ensure that the annotations accurately reflect our definition of cryptex code.

1) **Filtering Ineligible Projects:** Our process commences with the exclusion of projects that lack a diverse mix of both cryptex and non-cryptex functions.
2) **Identification of Cryptographic Functions:** We then systematically identify all instances of calls to cryptographic libraries or custom cryptographic implementations within the selected projects. These functions serve as initial indicators of cryptex code.
3) **Establishing an Initial Set:** Not all calls to cryptographic libraries are cryptex code, such as those related to error-handling functions. These non-sensitive calls are excluded, focusing instead on functions that implement cryptographic operations, which form the initial set of cryptex codeto be further refined in subsequent analytical phases.
4) **Snowballing:** We engage in a recursive exploration of the functions that either invoke or are invoked by these cryptex functions. *Backward Snowballing* examines the functions called by cryptex functions, as they are often cryptex code themselves. *Forward Snowballing* analyzes the functions that invoke cryptex functions, evaluating how they handle sensitive data to ascertain if they warrant designation as cryptex code.
5) **Global Variable Analysis:** In certain instances, global variables may store cryptex data. We systematically track the utilization of such variables and, subsequently, mark any functions that interact with them as cryptex when appropriate.

2) Phase 2 – Data Preprocessing: The task of identifying cryptex code is formulated as a binary sequence classification problem, with the Cryptex code and non-Cryptex Code classes. To achieve fine-grained, line-level classification, each line of code is treated as an independent prediction unit and passed to the LLM individually.

However, as discussed in CH3 (Long-Range Semantics), the semantics of a single line of code, especially when assessing its security relevance, cannot be fully understood in isolation. Both its local surroundings and its broader functional context are often critical for accurate interpretation. To this end, our data preprocessing pipeline constructs enriched input sequences that embed each target line within a broader semantic context.

Each input sequence is composed as:

$$\mathbf{x}_i = \begin{pmatrix} \texttt{code line} \\ +\,\texttt{local context} \\ +\,\texttt{global context} \\ +\,\texttt{metadata} \end{pmatrix}$$

During fine-tuning, each sample is defined as a tuple $(\mathbf{x}_i, \mathbf{y}_i)$, where $\mathbf{x}_i$ is the enriched input sequence and $\mathbf{y}_i \in \{0, 1\}$

is the binary ground truth label derived from the manual annotations in section IV-B1. A label of 1 indicates that the line is cryptex code, whereas 0 denotes non-cryptex code. In the following, we explain how each of these components is extracted and encoded for input into the LLM.

Local Context Features. The semantics of a code line is often tied to its immediate context, and analyzing it in isolation may lead to incorrect interpretations. To address this, we construct a **local context window** around the target line, specifically the two preceding and two succeeding lines of code. This 5-line region provides the model with immediate control flow and the semantic cues necessary for accurate classification. During preprocessing, we exclude non-informative or untrainable lines from being selected as classification targets. Specifically, include statements, preprocessing macros, empty lines, comments, and lines that contain only braces. While some of these lines (e.g., include statements) may be relevant, their influence is better captured through global features. Filtering in this way ensures that only semantically meaningful and actionable lines are passed to the model for classification.

Global Semantic Features. As highlighted in CH3 (Long-Range Semantics), many cryptex code behaviors arise from long-range dependencies, particularly data flows that span multiple functions or even modules of the program. Capturing these relationships is essential for identifying cryptex code that cannot be understood from the local context alone. To incorporate such information, we construct a Function Call Graph (FCG) for each C program. The FCG encodes caller-callee relationships, enabling a broader understanding of how different functions interact. For every line of code, we identify the enclosing function and extract the API calls and internal function invocations it contains. This list is added to the input sequence associated with that line to provide function-level semantic context, which showcases the input line's role within the broader program logic.

To further augment global semantic features, we additionally incorporate data flow features extracted from the Code Property Graph (CPG) [39] of each source file. For every line of code, we analyze the underlying CPG structure to identify relevant node types (e.g., `IDENTIFIER` `METHOD_PARAMETER_IN`) and track data dependencies via edges such as `REACHING_DEF`, `CFG`, and `AST`. This allows us to statically pre-compute and annotate each line with information such as the variables it defines or uses, and the set of lines it semantically reaches. These features capture subtle, long-range data flows that are key to recognizing security sensitive behavior. The background on the code property graphs is available in the Appendix D.

Metadata Information. To enhance the structural and contextual understanding of each code line, we incorporate metadata features such as the line number, function name, and file name directly into the input sequence. These attributes help the model disambiguate code lines that may be syntactically similar but occur in distinct locations or functional contexts within the program. For instance, the same API call might appear in both utility functions and security-critical routines, each carrying different implications.

3) Phase 3 - LLM Fine-tuning: As discussed in Section III-B and CH1 (Cryptex Code-Definition), static analysis alone is insufficient to detect cryptex functions. To automate this identification, we fine-tune a pre-trained LLM using the PEFT approach QLoRA [24]. QLoRA, rather than updating all model parameters, inserts lightweight adapters at key points in each transformer block. Prior to training, each enriched input sequence $\mathbf{x}_i$ is tokenized with the model's native subword tokenizer. The fine-tuned model then processes each sequence and outputs a single logit that is converted via a sigmoid function to yield a probability for the security sensitivity of the corresponding code line.

QLoRA Adapter Configuration: QLoRA has two main configurable aspects: The *rank* of the low-rank trainable matrices and the *number of adapter modules* placed within each transformer block. The rank determines the expressive capacity of each adapter, with higher values allowing more flexibility at the cost of increasing the number of parameters. The number and placement of adapters, on the other hand, directly impact how much of the attention mechanism can be adapted to the downstream task.

Modern LLMs use stacks of transformer blocks, each containing Multi-Head Self-Attention (MHSA) layers. We target the three key projection matrices ($\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$) within the MHSA layers because they govern the attention mechanisms critical for capturing task-specific semantics. Moreover, we also include an adapter at the output projection matrix $\mathbf{W}_O$ to better steer the aggregated attention outputs towards the domain task, which further improved downstream performance. The background of the key projection matrices can be found in the Appendix C. We configure QLoRA by adding four adapter modules per transformer block at these key projections.

In terms of rank configuration, we set the rank of the low-rank adapters to 16, striking a balance between parameter efficiency and task adaptability. This choice was empirically validated to provide sufficient expressive power for learning cryptex code patterns without incurring significant memory overhead during training.

Training Objective and Setup: As defined in IV-B2, each training instance is represented as a tuple, where during training, the model learns a function $(\mathbf{x}_i, \mathbf{y}_i)$,

$$f_\theta(\mathbf{x}_i) \rightarrow \hat{y}_i,$$

where θ represents the adapter weights introduced via QLoRA. The model outputs a logit $\hat{y}_i \in \mathbb{R}$, which is passed through a sigmoid activation [40] to generate the predicted probability. We use the Binary Cross-Entropy (BCE) loss [41], a well-established function for binary classification tasks, to train the model for distinguishing cryptex code from non-cryptex code lines. **Handling Class Imbalance:** Since the dataset is dominated by non-cryptex code lines, we employ a weighted BCE loss that assigns higher importance to the cryptex samples.

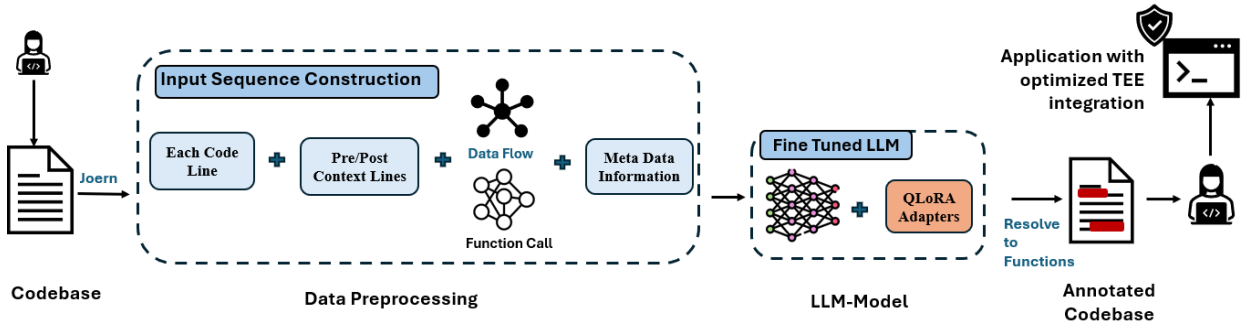

Fig. 2: LLM-CAL Inference Workflow

This approach scales the contribution of the minority class and stabilizes gradient updates during fine-tuning.

Optimization and Stability Configurations: To further ensure efficient and stable fine-tuning, we incorporate several best practices:

- *Weight Decay:* Acts as a regularizer to prevent overfitting by penalizing large weight updates.
- *Cosine Learning Scheduler:* Gradually warms up the learning rate, then decays it smoothly to help the optimizer converge effectively.
- *Early Stopping:* Monitors validation performance to halt training when no further improvements are observed, thereby conserving computational resources.

These strategies, combined with our QLoRA-based fine-tuning approach, allow the model to efficiently learn to identify cryptex code patterns despite the challenges of imbalanced and low-resource datasets.

4) Phase 4 – Inference: In the final phase, we deploy the fine-tuned LLM in conjunction with the preprocessing pipeline from Phase 2 (cf. Section IV-B2) to automatically annotate cryptex lines in unseen C codebases. The end-to-end inference workflow is illustrated in Figure 2. The process begins with developers supplying raw C source files in plaintext format. For each file, our inference module constructs enriched input sequences for all semantically relevant lines, filtering out non-informative lines such as comments, includes, braces, and empty lines. Using the same procedures as during training, the preprocessing module extracts the local context window, global semantic features, and metadata. Each resulting input sequence is tokenized and passed through the fine-tuned LLM with QLoRA adapters. For every input line, the model produces a scalar output between 0 and 1, representing the predicted probability that the corresponding code line is cryptex code. Lines with predicted scores above a predefined threshold such as 0.5 are flagged as cryptex code. This line-level prediction is then resolved to a function-level prediction. To achieve this, we statically assign each code line to its enclosing function using a lightweight parsing step. This automated annotation enables developers to efficiently identify and isolate critical code regions for TEE partitioning, ultimately reducing the application's trusted computing base and minimizing its attack surface.

V. Evaluation

This section evaluates LLM-CAL in identifying cryptex code at the line and function levels, emphasizing high recall and accuracy. We describe the dataset, experimental setup, and evaluation metrics. The evaluation also covers three case studies including assessment of LLM-CAL's performance on real world bitcoin utility tool, firmware code from TZ-DATASHIELD [21], out-of-distribution cryptographic code from alternative library. A comprehensive robustness evaluation of LLM-CAL and an ablation study detailing the impact of each component of LLM-CAL are provided in Appendices E and B.

A. Dataset

To create a robust foundation for training and evaluating the LLM-CAL model, we collected and constructed a first-of-its-kind, line- and function-level dataset following the guideline process outlined in Section IV-B1. To provide key insights into the dataset, this section covers aspects such as its coverage, and the effort involved while also detailing the de-duplication techniques used to ensure a clean and high-quality dataset. The details about the dataset size are provided in Appendix A.

Coverage: We have collected 4010 source code files from 1070 open-source software projects written in C from GitHub, each utilizing the OpenSSL [42] library. Specifically, these projects cover various categories, including cryptographic algorithms and implementations, OpenSSL integration and extensions, security tools and libraries, networking, and secure communication, as well as application development with platform-specific implementations. The projects, therefore, interact with the OpenSSL in diverse ways, which ensures that LLM-CAL trained on our dataset can generalize effectively and be applied to arbitrary projects using the library. The largest project in our dataset contains $13,428$ lines of code, while the median size across all projects is 492 lines of code.

OpenSSL was selected as the underlying cryptographic library for our dataset as it is one of the most widely used libraries for secure communications, with extensive deployment across a broad spectrum of applications and platforms. This widespread usage provides a rich and varied dataset, capturing numerous interaction patterns and edge cases with a well-established cryptographic standard. While OpenSSL serves as

the primary source for training due to its widespread use and rich interaction patterns, our objective is to capture the core semantics of cryptex code. This allows the model to generalize beyond OpenSSL and remain effective across different cryptographic libraries and real-world domains, including embedded firmware.

Effort: The dataset construction required substantial manual expert annotation effort, carried out by a dedicated team of students and researchers, with guidance from industry experts. More than 1000 hours were spent labeling cryptex codesections across all projects, ensuring accuracy and consistency throughout the dataset.

Dataset Split: The curated dataset was divided into an 80% training/validation set and a 20% independent hold-out test set to ensure unbiased evaluation.

B. Experimental Setup and Evaluation Metrics

This section presents the experimental setup, training strategy, and evaluation metrics used in our approach to assess the performance of LLM-CAL.

Quantization: We apply 4-bit quantization to the pre-trained Gemma-2B model using the QLoRA framework, significantly lowering GPU memory usage while preserving full model capacity. This allows us to fine-tune efficiently on resource-constrained hardware, as only the lightweight adapter layers remain in higher precision and are updated during training.

Tokenization: Tokenization serves as the bridge between structured input sequences and the LLM by converting textual inputs into numerical token IDs that the model can process. We use the tokenizer from the Gemma-2B model to encode each enriched sequence comprising local context, global function-level signals, and metadata into a fixed-length format. This step is applied consistently across training and inference to ensure compatibility with the model's embedding space.

Hardware: Training and inference were performed on a high-performance Linux server equipped with an Intel(R) Xeon(R) CPU and 251 GB of system memory. The experiments leveraged four GPUs, each with 48 GB of dedicated VRAM, to enable efficient large-scale fine-tuning and inference with quantized large language models. The system was configured with CUDA and cuDNN to ensure optimal GPU utilization and parallel training support.

Distributed Training: To accelerate training and scale to larger batch sizes, we employ distributed data parallelism using four GPUs. The training workload is partitioned across devices such that forward and backward passes are computed independently on each GPU with synchronized gradient updates. This setup enables efficient utilization of available memory and compute, reducing training time while preserving convergence stability.

Evaluation Metrics: To comprehensively assess the performance for LLM-CAL, we evaluate its ability to classify individual code lines as either cryptex code or not.

By comparing the predicted classifications with expert-labeled data produced in accordance with guidelines described in Section IV-B1, we construct a confusion matrix [43] detailing True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).

- **Accuracy** is the proportion of correctly identified lines, calculated as $\frac{TP+TN}{TP+FP+TN+FN}$.
- **Recall** is the proportion of correctly identified cryptex codelines, calculated as $\frac{TP}{TP+FN}$.
- **Precision** is the proportion of identified cryptex codelines that are actually cryptex code, calculated as $\frac{TP}{TP+FP}$.
- **F1 Score**: A harmonic mean that combines precision and recall, calculated as $\frac{2*(P*R)}{P+R}$.
- **TCB Reduction** is the percentage of lines not predicted as secure-sensitive per source file, reflecting how much of the code can be excluded from the TCB based on LLM-CAL's predictions. This value is compared with most optimal TCB reduction value computed from ground truth annotations.
- **Identification Rate**: The proportion of actual cryptex codelines or functions correctly identified by LLM-CAL, calculated as $\frac{\text{Identified Cryptex}Lines}{\text{Total Cryptex}Lines}$.

In our context, the LLM-CAL model places a higher emphasis on achieving a high recall, as it is crucial to correctly identify cryptex regions. Even in cases where a few cryptex lines are missed, we ensure that the corresponding cryptex functions are still correctly identified. Moreover, it is also important for LLM-CAL not to significantly compromise on precision, minimizing false positives. Ultimately, the goal is for LLM-CAL to maintain a well balanced performance, effectively identifying cryptex codelines while ensuring fairness and accuracy across all line classifications. Additionally, the identification rate metric highlights LLM-CAL's ability to accurately identify cryptex codelines, which is crucial for minimizing missed detections and maintaining comprehensive coverage of cryptex codecomponents.

C. LLM-CAL Performance at Different Granularity

In this section, we present the performance of LLM-CAL at line-level and function-level granularity to demonstrate its ability to identify cryptex code effectively. While LLM-CAL performs well at both levels, line-level probability scores offer fine-grained accuracy and allow us to pinpoint cryptex codelines across entire code files. Achieving high performance at this granularity is inherently more challenging due to the complexity and details involved in learning localized code patterns. At the same time, moving entire functions to TEEs is a more practical and efficient solution in real-world scenarios, as it avoids the overhead of frequent transitions between TEE and non-TEE environments. Therefore, identifying all cryptex code at the function level is equally critical. Together, these evaluations demonstrate that LLM-CAL not only performs well at detecting fine-grained cryptex code lines but also offers actionable guidance for secure TEE partitioning at the function scope.

1) LLM-CAL Performance at Line Level: The performance of LLM-CAL in identifying the total number of

LLM Model	Accuracy	Precision	Recall	F1-Score
Gemma-2B	99.04%	99.40%	97.50%	98.41%

TABLE I: LLM-CAL's performance on Test Set

cryptex codelines that can be annotated automatically from the custom dataset is presented in Tables I and II.

LLM-CAL demonstrates strong performance in identifying cryptex code lines across a diverse set of unseen C codebases. As shown in Table I, LLM-CAL achieves an overall accuracy of 99.04%, a precision of 99.40%, and a recall of 97.50% at the line level. The high recall highlights LLM CAL's robustness in minimizing missed detections of cryptex code, ensuring comprehensive coverage even in subtle or obfuscated scenarios. While the recall emphasizes the model's ability to capture nearly all relevant lines, the precision of 99.40% showcases the model's very minimal false positive rate and its ability to rightly differentiate between cryptex code and non-cryptex code lines without leading to unnecessary overprotection of the code. Further in Table II, we showcase the distribution of true and false predictions. LLM-CAL achieves a very high true positive rate, identifying $17,606$ cryptex codelines with very few 452 false negative lines. However, as all the corresponding cryptex functions of these lines are identified rightly, as shown in Table II, this minimal miss detection rate has no damage. This zero miss rate at function level is critical in secure code offloading scenarios where undetected sensitive operations can compromise the integrity of Trusted Execution Environments Importantly, the model also achieves a very low false positive count (106 out of 96,080 total lines), demonstrating that LLM-CAL balances between comprehensive coverage and very minimal over prediction.

• **TCB Reduction**: From Table II, it is evident that the LLM-CAL model achieved very low false positive and false negative rates, resulting in a significant reduction in TCB size, with only a 0.59% inflation due to misclassifications. Additionally, the influence of false negatives on TCB was minimal, with only 2.5% more reduction possible if they were fully eliminated.

2) LLM-CAL Performance at Function Level: The function-level analysis aims to evaluate LLM-CAL's ability to detect entire functions as security-sensitive based on line-level predictions. By automatically identifying and flagging these entire functions as cryptex, we help the developers in determining which functions should be relocated to TEE. Therefore, the performance of LLM-CAL at the function level is equally critical. As shown in Table II, LLM-CAL successfully identifies all the 684 security-sensitive functions and 3755 non-sensitive functions in the test set, achieving a 100% identification rate. This result clearly demonstrates that LLM-CAL has learned the patterns of cryptex and non-cryptex code rightly and is delivering accurate annotations to assist developers in TEE migration. Overall, LLM-CAL represents an advancement over state-of-the-art tools such as Graphene [17], Haven [18], and SCONE [19], which migrate entire applications into TEEs without fine-grained code sensitivity analysis.

Metric	Line-Level	Function-Level
True Positives (TP)	17,606	684
True Negatives (TN)	77,916	3755
False Positives (FP)	106	0
False Negatives (FN)	452	0
Total Predictions: cryptex	17,712	684
Total Predictions: non-cryptex	78,368	3755
Total Ground Truth: cryptex	18,058	684
Total Ground Truth: non-cryptex	78,022	3755
Total	96,080	4439

TABLE II: Comparison of line-level and function-level prediction summary

D. Case Study 1: Bitcoin Utility

To showcase the capabilities of the LLM-CAL in real-world projects, we examine a Bitcoin utility tool called *btcsigning*. This tool is a simplified version of the Bitcoin CLI tool provided by libbtc [44]. Both btcsigning and libbtc use OpenSSL along with its extension, libsecp256k1, for ECDSA operations on the secp256k1 elliptic curve.

The *btcsigning* tool has access to a transaction signing key in Wallet Import Format (WIF), which allows it to sign arbitrary messages using that key, including genuine Bitcoin transactions. The exposure of such a key could lead to substantial financial losses, emphasizing the importance of protecting it securely via TEE.

Line	Code	Prob.
1	`unsigned char private_key_hex[32];`	0.8322
2	`if (decode_wif_to_hex(wif_key, private_key_hex)!= 32){`	0.8092
3	`  printf("Invalid WIF key\n");`	0.7310
4	`  return 1;`	0.8300
5	`char un_base58_address[34];`	0.8187
6	`bitcoin_getaddress(un_base58_address , private_key_hex);`	0.9854

TABLE III: Converting signing key

Table III shows the line number, code, and probability (orange for cryptex and blue for non-cryptex) from LLM-CAL, where the private signing key is decoded from WIF format to raw binary format that is used by libsecp256k1, and the raw key is written into unsigned char array $private\_key\_hex$. Next, this private key is used to derive the Bitcoin address associated with the private key. The right column of the table shows the probabilities produced by LLM-CAL for these cryptex code lines. As we can see from the listing, LLM-CAL correctly predicts that the code lines are cryptex with above 80% probability for all.

Once the Bitcoin address is generated, the next crucial step involves signing the message using the $private\_key\_hex$, as shown in Table IV. In this step, the private key is used to sign the message's hash value. LLM-CAL correctly assesses the probability as high (99.16%) in line 2. However, as indicated in the table, LLM-CAL mistakenly evaluates *print* of the actual signature value as cryptex code (cf. line 14),

Line	Code	Prob.
1	`unsigned char bitcoin_sig[65];`	0.9697
2	`int bitcoin_sig_len = bitcoin_sign( bitcoin_sig, hash2, private_key_hex) ;`	0.9916
3	`if (bitcoin_sig_len < 0){`	0.9896
4	`printf("Error signing the message\n ");`	0.9407
5	`return 1;`	0.9196
6	`}`	
7	`printf("bitcoin_sig: ");`	0.7826
8	`for (int i = 0; i < 65; i++){`	0.7994
9	`printf("%02x", bitcoin_sig[i]);`	0.9196
10	`}`	
11	`printf("\n");`	0.8438
12	`//Encode the signature in base64`	
13	`char *signature_base64 = base64_encode(bitcoin_sig, bitcoin_sig_len);`	0.9796
14	`printf("signature_base64: %s\n", signature_base64);`	0.9914

TABLE IV: Signing operation

even though this value is public in the Bitcoin context. In other scenarios, such as when a verifier may accept the same signature multiple times, the signature value could be considered secret. This does not apply to Bitcoin, where the system is designed to prevent double-spending. Therefore, it is reasonable that LLM-CAL, not being trained to understand the specific logic of Bitcoin, tends to classify code as cryptex code more aggressively due to the requirements of other use cases.

Line	Code	Prob.
1	`void double_sha256(const unsigned char *input, size_t length, unsigned char *output){`	0.9648
2	`unsigned char hash[ SHA256_DIGEST_LENGTH];`	0.8449
3	`SHA256(input, length, hash);`	0.9875
4	`SHA256(hash, SHA256_DIGEST_LENGTH, output);`	0.9899
5	`}`	0.2814

TABLE V: Hashing Operation Function

Table V demonstrates that LLM-CAL exhibits good precision, correctly assigning higher probability scores to hash operations which aligns with the cryptex code definition outlined in Section IV-B1. Moreover, LLM-CAL assigns a lower probability to line 5, as the line content remains non-cryptex.

E. Use Case 2: Firmware Code Analysis

While LLM-CAL is primarily fine-tuned on a dataset with standard C programs that include OpenSSL based cryptographic operations, we further evaluate its robustness on a different domain: Low-level embedded firmware. For this, we analyze the *Pinlock* firmware project from the TZ-DATASHIELD dataset [45], which features hardware interfacing code and developer defined annotations for protecting sensitive data in systems built on ARM TrustZone architecture. The **Pinlock** application implements a simple authentication workflow that validates a user-entered PIN by computing its SHA-256 hash and comparing it to a stored reference key. If the hash matches, the system unlocks and grants access; otherwise, it remains locked, emulating secure access control mechanisms common in embedded systems. The source code uses the crypto library by NXP [46] instead of OpenSSL, showing the applicability of LLM-CAL to other crypto libraries. NXP offers its cryptographic library freely and is available in the MCUXpresso SDK [47].

The **Pinlock** firmware begins by declaring a set of critical variables required for input handling and system feedback, as shown in Table VI. LLM-CAL assigns high confidence to lines involving buffer and flag declarations specifically, the buffer used for PIN input storage and the pin received flag, both marked with TrustZone Data Section (TZDS) annotations in the original codebase. This highlights the model's ability to recognize sensitive data handling even *without explicit developer annotations*. Further, Table VII captures the key

Line	Code	Prob.
1	`uint8_t fmt_buffer[32];`	0.5084
2	`uint8_t buffer[BUFFER_SIZE] = {0};`	0.7416
3	`volatile uint32_t rx_index = 0;`	0.2999
4	`volatile bool pin_received = false;`	0.2227

TABLE VI: Declaration of buffers and variables

comparison logic used to authenticate the user-entered PIN against the stored reference in line 2. LLM-CAL confidently flags the memcmp-based comparison and return logic as cryptex code, correctly identifying this operation as central to access control.

Line	Code	Prob.
1	`bool match(uint8_t *key_received)`	0.5655
2	`if (memcmp(key_received, key_stored, 32)== 0){`	0.7065
3	`return true;}`	0.4167
4	`return false;`	0.6206

TABLE VII: Key Match Operation

Table VIII illustrates the hash computation process used to transform the entered PIN into a fixed-length representation using SHA-256. LLM-CAL accurately highlights the cryptex code lines involving interrupt disabling, hash invocation, and output size validation, demonstrating its ability to detect sensitive transformations in firmware logic as well. Despite lacking such annotations during inference, LLM-CAL

Line	Code	Prob.
1	`PRINTF("You entered: %s\r\n", buffer );`	0.4893
2	`DisableIRQ(DEBUG_USART_FLEXCOMM_IRQN );`	0.5382
3	`status = HASHCRYPT_SHA(HASHCRYPT, kHASHCRYPT_Sha256, buffer, rx_index,`	0.4629
4	`key_received, &output_size);`	0.6495
5	`assert(kStatus_Success == status);`	0.6583
6	`assert(output_size == 32u);`	0.6522

TABLE VIII: Hashing Operation

successfully identified key cryptex operations such as declaration of sensitive variables, key match, and hashing operations. This demonstrates LLM-CAL 's ability to generalize to code

for embedded devices that use different crypto libraries and effectively adapt to real-world codebases.

F. Usecase 3: Evaluating LLM-CAL on Out-of-Distribution Crypto Code

To further evaluate the generalizability of LLM-CAL beyond OpenSSL-based C programs, we apply it to a cryptographic code snippet using the mbedTLS [48] library. Specifically, we analyze a C source file obtained from an open-source project [49], which demonstrates AES-GCM encryption and decryption, where sensitive operations such as key and IV generation, encryption with authenticated data, and decryption with tag verification are implemented using the mbedTLS API. This example allows us to explore how LLM-CAL performs on an entirely different cryptographic library, without having seen such syntax or API calls during training. The code begins by initializing entropy and random number generator contexts, followed by seeding the CTR-DRBG component, as seen in Table IX. Next, it generates cryptographic key and Initialization Vector (IV) values, configures the GCM context, and performs authenticated encryption and decryption using *mbedtls_gcm_crypt_and_tag* and *mbedtls_gcm_auth_decrypt*. These steps are essential for securing messages in real-world embedded and networked systems.

Line	Code	Prob.
1	`mbedtls_entropy_init(&entropy);`	0.8757
2	`mbedtls_ctr_drbg_init(&ctr_drbg);`	0.8714
3	`mbedtls_gcm_init(&gcm);`	0.6315
4	`ret = mbedtls_ctr_drbg_seed(& ctr_drbg, mbedtls_entropy_func, & entropy, (unsigned char *)pers, strlen(pers));`	0.2509
5	`if( ret != 0 ){`	0.5308
6	`printf("mbedtls_ctr_drbg_seed() failed - returned -0x\%04x\n", -ret) ;`	0.4358
7	`goto exit;}`	0.098

TABLE IX: Entropy and randomness initialization

Tables IX, X, and XI showcase sensitive blocks from this source file along with LLM-CAL 's predicted probabilities for each line. In Table IX, we observe that LLM-CAL assigns high confidence to lines 1 through 6, which are responsible for entropy pool setup and seeding, correctly identifying them as cryptex code. In Table X, the actual AES-GCM encryption operation receives the highest probability, with supporting lines like error handling and ciphertext generation also marked as cryptex code. Similarly, Table XI highlights LLM-CAL's ability to detect the authenticated decryption logic with high confidence, while appropriately assigning a low score to the final print statement that simply logs the plaintext.

These results demonstrate that LLM-CAL is capable of adapting to new and alternate cryptographic APIs and function calls not present during fine-tuning, making robust predictions on previously unseen secure operations, even across library boundaries.

Line	Code	Prob.
1	`ret = mbedtls_gcm_crypt_and_tag(&gcm , MBEDTLS_GCM_ENCRYPT, plain_len, iv , IV_BYTES, add_data, add_len, input , output, TAG_BYTES, tag);`	0.8210
2	`if( ret != 0 ){`	0.7799
3	`printf("mbedtls_gcm_crypt_and_tag failed to encrypt the data - returned -0x\%04x\n", -ret);`	0.8031
4	`goto exit;}`	0.3645
5	`printf("ciphertext: '%s' (length \% zu)\n", output, strlen((char*)output ));`	0.4850

TABLE X: GCM encryption with tag generation

Line	Code	Prob.
1	`ret = mbedtls_gcm_auth_decrypt(&gcm , plain_len, iv, IV_BYTES, add_data , add_len, tag, TAG_BYTES, output, decrypted);`	0.6934
2	`if( ret != 0 ){`	0.9334
3	`printf("mbedtls_gcm_auth_decrypt failed to decrypt the ciphertext - tag doesn't match\n");`	0.8128
4	`goto exit;}`	0.5751
5	`printf("decrypted : '%s' (length \% zu)\n", decrypted, strlen((char *) decrypted));`	0.5547

TABLE XI: GCM decryption and authentication

G. LLM-CAL Robustness

This section evaluates the robustness of LLM-CAL by analyzing its performance under varying conditions, including different fine-tuning strategies, diverse LLM architectures, and comprehensive evaluation on alternative library.

*1) **Comparison of Fine-Tuning Strategies**:* To assess the effectiveness of different fine-tuning strategies, we compare LLM-CAL's performance across three configurations: the base Gemma-2B pretrained model used in zero-shot mode which means there is no task adaptation and fine tuning on the model, QLoRA-based fine-tuning with 4-bit quantization and adapter injection, and DORA-based [30] fine-tuning using dynamic offset rank adaptation. Each configuration is evaluated at the line level using the same dataset and inference pipeline.

Fine-Tuning Method	Accuracy	Precision	Recall	F1-Score
Zero-Shot (Gemma-2B)	83.02%	90.11%	54.09%	52.85%
QLoRA (with LLM-CAL)	99.04%	99.40%	97.50%	98.41%
DORA (with LLM-CAL)	91.94%	84.86%	95.06%	88.43%

TABLE XII: Comparison of LLMCAL fine-tuning strategies on line-level performance.

As observed in Table XII, the zero-shot setting performs poorly on the target task, despite achieving high precision. The model fails to generalize to security-sensitive codes, resulting in a low recall (54.09%) and an F1-score of just 52.85%. This emphasizes the limitations of using general-purpose LLMs without adaptation for cryptex code identification.

In contrast, both QLoRA and DORA fine-tuning pipelines show substantial improvements across all metrics. QLoRA

with best suited training hyperparameters outperforms DORA, demonstrating better alignment with the distribution of cryptex code lines. These results showcase that domain-specific fine-tuning is essential for high-fidelity cryptex code annotation. The comprehensive evaluation on alternative crypto library and diverse llm architectures are provided in appendix E.

VI. Related Work

This section provides an overview of related literature. Notably, there is no directly relevant work addressing the specific problem that LLM-CAL aims to solve. We discuss the most closely related work to LLM-CAL, specifically literature that focuses on simplifying and automating the migration of applications to TEEs, RNN and LLM-based vulnerability detection methods.

Tooling for TEE Migrations Tools like Graphene [17], Haven [50], and SCONE [51] enable the migration of entire software environments into secure enclaves. Graphene and Haven facilitate legacy applications, while SCONE targets containerized microservices. Although these tools simplify TEE integration, they focus on full migration rather than minimizing the TCB, contrasting with LLM-CAL's approach.

Other tools have explored optimizing TEE boundaries, similar to LLM-CAL, but require significant developer interaction. TEE-DRUP [22] helps developers partition applications by in-sourcing security-sensitive variables based on natural language heuristics, yet its reliance on manual curation makes it less practical. SOAAP [20] targets multi-compartment technologies like CHERI [52] and relies heavily on developer-provided annotations, making it time-consuming compared to the fully automated approach of LLM-CAL that works directly on unannotated source code. Similarly, a compiler toolchain TZ-DATASHIELD [21] enables secure and efficient protection of sensitive data in ARM TrustZone-enabled microcontrollers by analyzing annotated source code to generate fine-grained, data-flow-aware compartments. Again, unlike LLM-CAL, TZ-DATASHIELD relies on developer-annotated source code.

RNN-based Vulnerability Detection Between 2018 and 2021, Recurrent Neural Networks (RNNs) were widely explored for source code vulnerability detection, typically by vectorizing code and processing it with bidirectional RNN architectures. VulDeePecker [53] introduced the concept of "code gadgets" and employed Bi-LSTMs to process semantically related code lines, contributing a specialized dataset and key principles for vulnerability detection. SySeVR [54] refined this by using BGRUs and introducing Syntax-based and Semantics-based Vulnerability Candidates (SyVCs and SeVCs) that incorporate syntax and data/control flow dependencies. VulDeeLocator [55] extended SySeVR by integrating intermediate code for richer semantic context and introduced BRNN-vdl, which combines attention and granularity refinement. Tang *et al.* [56] compared Bi-LSTMs and RVFLs within the VulDeePecker framework and showed that Bi-LSTM with doc2vec preprocessing achieved the best results.

LLM-based Vulnerability Detection Recent research has focused on using LLMs for vulnerability detection in both source and binary code. While LLMs achieve high recall, they often suffer from low precision, leading to false positives [57], [58]. Pearce *et al.* [59] assessed five LLMs using CodeQL bug reports and found that while all bugs were fixed, generating functionally correct code remained challenging. DefectHunter [60] enhanced Transformers with Conformers [61] and used Abstract Syntax Tree's, Control Flow Graph's, and Data Flow Graph's alongside LLMs to capture structural code features. Chan *et al.* [62] compared zero/few-shot learning and fine-tuning, showing fine-tuning as most effective despite frequent update needs. Liu *et al.* [57] used LLMs to automate manual tasks in binary taint analysis, identifying sensitive functions and tracing data dependencies. Luo *et al.* [58] introduced VulHawk, combining RoBERTa and GNNs for binary code search in IoT firmware, improving precision with an entropy-based adapter and progressive search. Chen *et al.* [63] evaluated GPT-3.5 and GPT-4 on smart contracts, noting high recall but inconsistent precision. Purba [64] compared LLMs to static analysis and neural networks across two public datasets, finding direct prompting ineffective and fine-tuning helpful, though high false positives persisted.

Summary In summary, LLM-CAL is pioneering in addressing the automation approach to identify cryptex code, an area not covered by current research, making it a foundational contribution to the fields of TEE migration and secure code analysis. None of the works in the three discussed categories are directly relevant or can be directly compared to LLM-CAL. First, while existing methods for automating application migration to TEEs share some objectives, they lack a focus on identifying cryptex code. Second, LLM-based vulnerability detection approaches, though similar in using LLMs for code analysis, diverge significantly in scope and methods from LLM-CAL's specialized focus.

VII. Conclusion

In summary, LLM-CAL offers a lightweight and scalable solution for identifying cryptexlines of code that focus on cryptographic logic and related data flows using large language models. By combining structured input construction with parameter-efficient fine-tuning via QLoRA, LLM-CAL effectively captures local, global, and semantic context. Our results show strong performance across diverse settings including OpenSSL code, embedded firmware, and alternate crypto programs with mbedTLS and NXP demonstrating LLM-CAL 's ability to generalize beyond its training distribution. This provides practical, automated, and developer-independent cryptex code annotations for real-world systems.

References

[1] S. Kim, J. Park, K. Lee, I. You, and K. Yim, "A brief survey on rootkit techniques in malicious codes." *J. Internet Serv. Inf. Secur.*, vol. 2, no. 3/4, pp. 134–147, 2012.

[2] "CVE-2024-39291 - Linux Kernel Overflow." [Online]. Available: https://www.cvedetails.com/cve/CVE-2024-39291/

[3] "CVE-2016-5195 - Dirty COW." [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2016-5195
[4] "CVE-2023-34322 - Xen PV Improper Check." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-34322
[5] "CVE-2023-4155 - KVM Race Condition." [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2023-4155
[6] "CVE-2024-3094 - Linux XZ Backdoor." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-3094
[7] "CWE-506 - Embedded Malicious Code." [Online]. Available: https://cwe.mitre.org/data/definitions/506.html
[8] "CWE-119 - Buffer Overflow." [Online]. Available: https://cwe.mitre.org/data/definitions/119.html
[9] "CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor." [Online]. Available: https://cwe.mitre.org/data/definitions/200.html
[10] "CWE-691 - Insufficient Control Flow Management." [Online]. Available: https://cwe.mitre.org/data/definitions/691.html
[11] "CWE-693 - Protection Mechanism Failure." [Online]. Available: https://cwe.mitre.org/data/definitions/693.html
[12] "CWE-94 - Code Injection." [Online]. Available: https://cwe.mitre.org/data/definitions/94.html
[13] "Intel Software Guard Extensions (SGX)." [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html
[14] "TrustZone for Cortex-M." [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-m
[15] S. Pinto and N. Santos, "Demystifying ARM TrustZone: A comprehensive survey," *ACM computing surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.
[16] "TrustZone for Cortex-A." [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a
[17] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 645–658.
[18] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.
[19] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "SCONE: Secure Linux containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 689–703.
[20] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with soaap," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1016–1031.
[21] Z. Kong, M. Park, L. Guan, N. Zhang, and C. H. Kim, "TZ-DATASHIELD: Automated Data Protection for Embedded Systems via Data-Flow-Based Compartmentalization," in *Proceedings of the 32nd Network and Distributed System Security Symposium (NDSS 2025)*, San Diego, CA, Feb. 2025. [Online]. Available: https://dx.doi.org/10.14722/ndss.2025.240563
[22] Y. Liu and E. Tilevich, "Reducing the price of protection: Identifying and migrating non-sensitive code in TEE," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020, pp. 112–120.
[23] D. Brown, *The Da Vinci Code*. Doubleday, 2003.
[24] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: efficient finetuning of quantized llms," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.
[25] G. Team and et al., "Gemma 2: Improving open language models at a practical size," *arXiv:2408.00118*, 2024.
[26] L. Team and et al., "The llama 3 herd of models," *arXiv:2407.21783*, 2024.
[27] C. Team and et al., "Codegemma: Open code models based on gemma," *arXiv:2406.11409*, 2024.
[28] Z. Fu, H. Yang, A. M.-C. So, W. Lam, L. Bing, and N. Collier, "On the Effectiveness of Parameter-Efficient Fine-Tuning," *arXiv:2211.15583*, November 2022.
[29] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv:2106.09685*, 2021.
[30] S.-Y. Liu, C.-Y. Wang, H. Yin, P. Molchanov, Y.-C. F. Wang, K.-T. Cheng, and M.-H. Chen, "Dora: weight-decomposed low-rank adaptation," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.
[31] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *In IEEE Symposium on Security and Privacy*, 2015.
[32] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, , and R. Strackx, "Telling Your Secrets without Page Faults: Stealthy Page Table Based Attacks on Enclaved Execution," in *26th USENIX Security Symposium*, 2017.
[33] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *USENIX Workshop on Offensive Technologies*, 2015.
[34] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.
[35] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu," in *ACM Symposium on Information, Computer and Communications Security*, 2017.
[36] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Network and Distributed System Security Symposium*, 2017.
[37] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.
[38] G. Team and et al., "Github," 2008. [Online]. Available: https://github.com/
[39] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.
[40] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
[41] K. P. Murphy, "Machine learning: A probabilistic perspective," *MIT press*, 2012.
[42] "OpenSSL - Cryptography and SSL/TLS Toolkit," 1998. [Online]. Available: https://www.openssl.org/
[43] "Glossary of terms," *Machine Learning*, vol. 30, no. 2, pp. 271–274, Feb 1998. [Online]. Available: https://doi.org/10.1023/A:1017181826899
[44] L. Team and et al., "libbtc," 2021. [Online]. Available: https://github.com/libbtc/libbtc
[45] Z. Kong, M. Park, L. Guan, N. Zhang, and C. H. Kim, "Artifacts for TZ-DATASHIELD," in *Network and Distributed System Security Symposium (NDSS) 2025*. San Diego, California: Zenodo, 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14257984
[46] "NXP website." [Online]. Available: https://www.nxp.com/
[47] NXP Semiconductors, "MCUXpresso Software Development Kit (SDK)," 2023. [Online]. Available: https://mcuxpresso.nxp.com/en/welcome
[48] TrustedFirmware.org, "Mbed tls," https://github.com/Mbed-TLS/mbedtls, 2006.
[49] T. Leonhardt, "Practical cryptography engineering," https://github.com/tleonhardt/practical_cryptography_engineering, 2018.
[50] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," *USENIX Symposium on Operating Systems Design and Implementation*, 2014.
[51] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux containers with Intel SGX," *USENIX Symposium on Operating Systems Design and Implementation*, 2016.
[52] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.
[53] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Network and Distributed Systems Security (NDSS) Symposium 2018*, Feb. 2018.

[54] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.
[55] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2821–2837, 2021.
[56] G. Tang, L. Meng, H. Wang, S. Ren, Q. Wang, L. Yang, and W. Cao, "A comparative study of neural network techniques for automatic software vulnerability detection," in *2020 International Symposium on theoretical aspects of software engineering (TASE)*. IEEE, 2020, pp. 1–8.
[57] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Li, and L. Sun, "Harnessing the power of llm to support binary taint analysis," *arXiv:2310.08275*, 2023.
[58] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *Proceedings of the 30th Network and Distributed System Security Symposium (NDSS 2023)*, San Diego, CA, Feb. 2023.
[59] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2339–2356.
[60] J. Wang, Z. Huang, H. Liu, N. Yang, and Y. Xiao, "Defecthunter: A novel llm-driven boosted-conformer-based code vulnerability detection mechanism," *arXiv:2309.15324*, 2023.
[61] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu *et al.*, "Conformer: Convolution-augmented transformer for speech recognition," in *Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH)*, 2020.
[62] A. Chan, A. Kharkar, R. Z. Moghaddam, Y. Mohylevskyy, A. Helyar, E. Kamal, M. Elkamhawy, and N. Sundaresan, "Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning?" *arXiv:2306.01754*, 2023.
[63] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *arXiv:2309.05520*, 2023.
[64] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2023, pp. 112–119.
[65] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
[66] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, p. 1–19, jul 1970. [Online]. Available: https://doi.org/10.1145/390013.808479
[67] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

APPENDIX

This appendix provides additional information on the data set and reports the results of the ablation study of LLM-CAL.

A. Dataset Split

The dataset was initially divided into an 80% training/validation set and a 20% independent hold-out test set to ensure unbiased evaluation. Table XIII shows the size of the hold-out test set and the training/validation split, including the number of cryptex code and non-cryptex source files.

Set	SCC Files	Non-SCC Files
Train+Validation Set	489	2912
Test Set	88	515

TABLE XIII: Dataset split by SCC and non-SCC file counts.

B. Ablation Study

In this section, we perform a detailed ablation study to evaluate the impact of various components and configurations on the performance of LLM-CAL. Using the best-performing LLM-CAL model as a baseline, our objective is to understand the contribution of various features and hyperparameters.

The Impact of Local Features. The inclusion of local context features significantly enhances the performance of LLM-CAL, as shown in Table XIV. When local features are added, the F1 score improves from 71.35% to 98.41%, and recall increases from 74.12% to 97.50%, demonstrating a stronger ability to correctly detect cryptex code lines. This improvement highlights the importance of surrounding code semantics captured via pre and post context lines in reducing false negatives and improving overall classification performance.

Ablation Setting	Accuracy	Precision	Recall	F1-Score
No Local Features	73.45%	68.82%	74.12%	71.35%
No Global Features	82.12%	76.24%	85.13%	80.44%
No Metadata Features	85.12%	82.20%	87.30%	84.62%
All Features (Full Model)	99.04%	99.40%	97.50%	98.41%

TABLE XIV: Ablation study showing the individual impact of local, global, and metadata features on LLM-CAL performance.

The Impact of Global Features. The inclusion of global semantic features has a positive impact on LLM-CAL's performance, as shown in Table XIV. When these features are excluded, LLM-CAL reduces to an F1 score of 80.44% and a recall of 85.13%. However, after incorporating global features such as function call graph and data flow dependencies the recall improves to 97.50%, indicating a stronger ability to capture security-sensitive lines. This enhancement also improved precision from 76.24% to 99.40%, suggesting quite a fewer false positives. Overall, the F1 score increases to 98.41%, and the accuracy reaches 99.04%, demonstrating that global features contribute to more balanced and accurate predictions in LLM-CAL.

The Impact of Metadata as Features. Metadata features, such as file name, function name, and line number, provide contextual information that help the model in disambiguating similarly structured code lines across different locations.

As shown in Table XIV, when meta data features are added, the primary impact is on recall, which increases from 87.30% to 97.50%.

C. Background: Transformer Block

Once the input tokens are converted into embeddings, they are passed through a stack of *Transformer blocks*, which form the core processing units in modern LLMs. Each Transformer block consists of two primary components:

- A Multi-Head Self-Attention (MHSA) mechanism, which captures contextual dependencies between tokens by computing attention scores over the sequence.
- A Position-wise Feedforward Network (FFN), which applies non-linear transformations independently to each token representation.

Additionally, each block is equipped with *residual connections* and *layer normalization* layers that help stabilize training and improve gradient flow. Formally, let $\mathbf{X} \in \mathbb{R}^{n \times d}$ represent the input token embeddings to a Transformer block, where n is the sequence length and d is the embedding dimension. The output of the block is computed as:

$$\mathbf{H} = \text{LayerNorm}\left(\mathbf{X} + \text{MHSA}(\mathbf{X})\right)$$

$$\text{Output} = \text{LayerNorm}\left(\mathbf{H} + \text{FFN}(\mathbf{H})\right)$$

Here, $\mathbf{H} \in \mathbb{R}^{n \times d}$ is the intermediate hidden state after the attention sub-layer. Stacking multiple such Transformer blocks allows the model to gradually build deep contextualized representations of the input sequence.

Multi-Head Attention Layer: The MHSA mechanism enables the model to jointly attend to information from different representation subspaces. It does so by projecting the input embeddings into multiple sets of *queries* ($\mathbf{Q}$), *keys* ($\mathbf{K}$), and *values* ($\mathbf{V}$) through learned linear transformations. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be the input to the MHSA layer. The projections are computed as:

$$\mathbf{Q} = \mathbf{X}W_Q, \quad \mathbf{K} = \mathbf{X}W_K, \quad \mathbf{V} = \mathbf{X}W_V$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}$ are learned weight matrices, and d_k is the dimensionality of each head. Each attention head computes scaled dot-product attention:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Outputs from all heads are concatenated and projected using another linear transformation W_O:

$$\text{MHSA}(\mathbf{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

These projection layers—W_Q, W_K, W_V, W_O—are the key locations where adapter modules are integrated in Parameter-Efficient Fine-Tuning approaches like LoRA and QLoRA.

D. Code Property Graph

Code Property Graphs (CPGs) [39] provide a unified graph-based representation of source code. They merge three standard program representations, which are Abstract Syntax Tree (AST) [65], Control Flow Graph (CFG) [66], and Program Dependence Graph (PDG) [67], into a single structure. This integration allows both syntactic and semantic aspects of code to be analyzed together.

The AST captures the syntactic structure of code, such as loops, conditions, and expressions. The CFG models all possible execution paths, while the PDG tracks how data and control flow between code elements. Combining these representations altogether provides a comprehensive reasoning about program behavior.

`IDENTIFIER` `METHOD_PARAMETER_IN`) and track data dependencies via edges such as `REACHING_DEF`, `CFG`, and `AST`.

Nodes in a CPG represent key code elements. Common types include `METHOD` (functions), `CALL` (function calls), `IDENTIFIER` (variables), and `METHOD_PARAMETER_IN`) (function inputs). These nodes are enriched with metadata like variable names, types, and code positions. Edges define relationships between nodes. `AST` edges represent syntactic structure, `CFG` edges capture control flow, and `REACHING_DEF` edges trace how variables are defined and used across the code. These edges are especially useful for analyzing data dependencies. In our work, we leverage these nodes and edges to extract data flow features for each line of code. This helps identify which variables are used or defined, and how they connect to other lines—crucial for recognizing cryptex code behavior.

E. LLM-CAL Robustness

1) ***Robustness Evaluation on Alternative Library****:* To further assess LLM-CAL's generalization beyond OpenSSL-based programs, we curated a separate test set consisting of **30** open-source projects that rely on the mbedTLS cryptographic library. Each project was manually reviewed and annotated with line-level and function-level cryptex labels, following the guideline process outlined in Section IV-B1. All C source files from these projects were used to evaluate LLM-CAL 's robustness on a previously unseen cryptographic codebase. This large-scale evaluation highlights LLM-CAL 's ability to remain accurate and consistent even when exposed to new cryptographic APIs, code patterns, and project structures not encountered during training.

Metric	Line-Level	Function-Level
True Positives (TP)	3,860	52
True Negatives (TN)	11,828	445
False Positives (FP)	12	0
False Negatives (FN)	96	0
Total Ground Truth: cryptex	3,956	52
Total Ground Truth: non-cryptex	11,840	445
Total	15,796	497

TABLE XV: LLM-CAL's prediction summary on mbedTLS-based test set

As shown in Table XV, the results confirm that LLM-CAL generalizes well across library boundaries and maintains its reliability in identifying cryptex operations at both line and function levels with zero false positives and negatives. The perfect scores on the mbedTLS set complement the already strong performance on the original test set. In Summary, this evaluation shows that LLM-CAL is not over-fitted to specific APIs seen during training, but instead learns meaningful semantic patterns that extend across cryptographic libraries.

2) ***Comparison Across LLM Architectures****:* We further examine the generalizability of the LLMCAL pipeline across different backbone models. Using the same QLoRA configuration, we fine-tune and evaluate three popular LLMs: Gemma-2B, CodeGemma-2B, and LLaMA-7B.

Table XVI indicates that LLM-CAL is adaptable across diverse LLM architectures. Larger model such as LLaMA-7B yield modest improvements in F1 score, but Gemma-2B offers

Model	Accuracy	Precision	Recall	F1-Score
Gemma-2B (Ours)	99.04%	99.40%	97.50%	98.41%
CodeGemma-2B	86.10%	61.49%	79.97%	69.59%
LLaMA-7B	84.23%	76.46%	85.46%	79.54%

TABLE XVI: LLM-CAL performance with different LLM backbones (line-level).

the best trade-off between performance and computational efficiency.

F. Runtime Performance of LLM-CAL

We assess the runtime efficiency of LLM-CAL by measuring the time required to process and annotate cryptex code lines in unseen C source files. This includes preprocessing steps such as context extraction, input sequence construction, tokenization, and model inference using the fine-tuned LLM. As shown in Table XVII, for a representative source file containing 352 lines, the total runtime was approximately 21.84 seconds on average. Tokenization and input sequence constructions take the largest portion of the runtime due to the overhead of the line-to-function mappings and structured metadata extraction. In contrast, LLM model loading and batched inference are much more efficient with only consuming approx 3.5 seconds, benefiting from GPU acceleration and model quantization. These results demonstrate that LLM-CAL is practical for real-world integration, offering substantially minimal time to produce high accurate annotations.

Source File	Tokenization + Preprocessing (s)	Model Loading (s)	Inference (s)	Total Time (s)
352 lines	18.35	2.91	0.58	21.84

TABLE XVII: Runtime performance of LLM-CAL on a representative source file

In summary, our evaluation demonstrates that LLM-CAL consistently identifies cryptex code across a diverse range of software codebases, including OpenSSL-based applications, embedded firmware, and mbedTLS-powered projects. Through quantitative metrics and qualitative case studies, we show that LLM-CAL generalizes well beyond its training distribution, effectively capturing both cryptographic operations and their surrounding semantics. Additionally, our runtime analysis confirms the feasibility of deploying LLM-CAL in practical developer workflows.

G. LLM-CAL Performance in Various Sensitive Operations

This subsection evaluates the adaptability of LLM-CAL in various cryptographic operations. Table XVIII summarizes the performance of LLM-CAL at the line level from our custom dataset across four core security-sensitive tasks such as encryption/decryption, hashing, key storage, and TLS-based communication. LLM-CAL demonstrates near-perfect performance in identifying cryptex code lines involving encryption, hashing, and secure key usage, achieving F1 scores of 1.00 with precision and recall above 0.99. These results highlight LLM-CAL's effectiveness in recognizing well-defined cryptographic routines with deterministic structural and semantic patterns. In TLS/DTLS-based secure communication, where cryptographic usage is often entangled with networking and I/O operations, LLM-CAL maintains a perfect recall of 1.00 while achieving an F1 score of 0.86. The relatively lower precision in this tasks 0.75 suggests some over prediction in complex, multi-functional code snippets yet the model reliably captures all true positives. Overall, the results ensure that LLM-CAL generalizes well across a range of distinct sensitive operations.

Sensitive Operation	LOC	Precision	Recall	F1	Accuracy
Encryption & Decryption	2530	0.99	1.00	1.00	0.998
Hashing & Digests	850	1.00	1.00	1.00	1.000
Key Storage & Secure Key Usage	6592	1.00	1.00	1.00	0.999
TLS / DTLS / Communication	7837	0.75	1.00	0.86	0.966

TABLE XVIII: LLM-CAL's performance on various cryptex operations.