CogniCrypt: Synergistic Directed Execution and LLM-Driven Analysis for Zero-Day AI-Generated Malware Detection

George Edward, Mahdi Eslamimehr

Quandary Peak Research

Abstract. The weaponization of Large Language Models (LLMs) for automated malware generation poses an existential threat to conventional detection paradigms. AI-generated malware exhibits polymorphic, metamorphic, and context-aware evasion capabilities that render signature-based and shallow heuristic defenses obsolete. This paper introduces **CogniCrypt**, a novel hybrid analysis framework that synergistically combines *concolic execution* with *LLM-augmented path prioritization* and *deep-learning-based vulnerability classification* to detect zero-day AI-generated malware with provable guarantees. We formalize the detection problem within a first-order temporal logic over program execution traces, define a lattice-theoretic abstraction for path constraint spaces, and prove both the *soundness* and *relative completeness* of our detection algorithm, assuming classifier correctness. The framework introduces three novel algorithms: (i) an LLM-guided concolic exploration strategy that reduces the average number of explored paths by 73.2% compared to depth-first search while maintaining equivalent malicious-path coverage; (ii) a transformer-based path-constraint classifier trained on symbolic execution traces; and (iii) a feedback loop that iteratively refines the LLM's prioritization policy using reinforcement learning from detection outcomes. We provide a comprehensive implementation built upon `angr` 9.2, `Z3` 4.12, Hugging Face `Transformers` 4.38, and `PyTorch` 2.2, with full configuration details enabling reproducibility. Experimental evaluation on the EMBER, Malimg, SOREL-20M, and a novel AI-Gen-Malware benchmark comprising 2,500 LLM-synthesized samples demonstrates that CogniCrypt achieves 98.7% accuracy on conventional malware and 97.5% accuracy on AI-generated threats, outperforming ClamAV, YARA, MalConv, and EMBER-GBDT baselines by margins of 8.4–52.2 percentage points on AI-generated samples.

Keywords: Concolic Execution, Large Language Models, AI-Generated Malware, Symbolic Execution, Vulnerability Discovery, Software Security, Formal Verification, Deep Learning, Zero-Day Detection, Secure Coding.

1 Introduction

The cybersecurity landscape is undergoing a fundamental transformation driven by the dual-use nature of Large Language Models (LLMs). While LLMs have accelerated legitimate software development through code generation, refactoring, and automated testing [1], adversaries have simultaneously exploited these capabilities to produce sophisticated malware at unprecedented scale and velocity [2, 3]. Recent threat intelligence reports document a 135% year-over-year increase in AI-assisted cyberattacks, with LLM-generated payloads exhibiting polymorphic behavior, semantic-level obfuscation, and adaptive evasion strategies that defeat traditional signature-based and static-heuristic defenses [4].

The fundamental challenge posed by AI-generated malware is threefold. First, LLMs can produce functionally equivalent but syntactically diverse variants of the same exploit, defeating hash-based and pattern-matching detectors. Second, AI-generated code can embed trigger conditions that activate malicious behavior only under specific environmental contexts, evading sandbox-based dynamic analysis. Third, LLMs can iteratively refine evasion strategies by analyzing detection feedback, creating an adversarial arms race that static defense postures cannot sustain.

Concolic execution,a portmanteau of *con*crete and symb*olic* execution,offers a principled approach to this challenge by systematically exploring program execution paths through the interplay of concrete test inputs and symbolic constraint solving [5, 6]. By maintaining both a concrete execution state and a symbolic path constraint, concolic engines can reason about the conditions under which specific program behaviors manifest, including latent malicious behaviors hidden behind opaque predicates and environmental checks. However, the well-known

David C. Wyld et al. (Eds): NLPML, CRIS – 2026
pp. 01-19, 2026. CS & IT - CSCP 2026
DOI: 10.5121/csit.2026.160901

path explosion problem,where the number of feasible paths grows exponentially with program size and branching complexity,has historically limited the scalability of concolic analysis for real-world malware detection [7, 8].

This paper introduces **CogniCrypt**, a framework that resolves the scalability limitation by employing an LLM as an intelligent *path oracle* that guides the concolic engine toward execution paths with high malicious potential. The key insight is that LLMs, having been pre-trained on vast corpora of source code and security advisories, possess an implicit model of "suspicious" program behavior that can be leveraged to prioritize the exploration of paths most likely to reveal malicious intent. CogniCrypt further incorporates a transformer-based *path constraint classifier* that maps symbolic execution traces to maliciousness scores, and a *reinforcement learning feedback loop* that continuously improves the LLM's prioritization policy based on detection outcomes.

Contributions. This paper makes the following contributions:

1. **Formal Framework:** We define a first-order temporal logic $\mathcal{L}_{\text{CogniCrypt}}$ over program execution traces and establish a lattice-theoretic abstraction of the path constraint space. We prove the *soundness* (no false negatives under the threat model) and *relative completeness* (detection of all malicious paths reachable within a bounded exploration budget) of the CogniCrypt detection algorithm (Section 3).
2. **Novel Algorithms:** We present three tightly integrated algorithms: LLM-Guided Concolic Exploration (Algorithm 1), Transformer-Based Path Constraint Classification (Algorithm 2), and Reinforcement-Learning-Based Policy Refinement (Algorithm 3) (Section 4).
3. **Comprehensive Implementation:** We provide a fully reproducible implementation built on `angr`, `Z3`, `PyTorch`, and Hugging Face `Transformers`, with detailed configuration, hyperparameter settings, and deployment instructions (Section 5).
4. **Extensive Evaluation:** We evaluate CogniCrypt on four benchmarks,EMBER [17], Malimg [18], SOREL-20M [19], and a novel AI-Gen-Malware dataset,demonstrating state-of-the-art performance, particularly on AI-generated threats (Section 6).

2 Related Work

2.1 Concolic and Symbolic Execution for Security Analysis

Symbolic execution was introduced by King [9] and has since become a cornerstone of program analysis. DART [6] and CUTE [5] pioneered concolic (dynamic symbolic) execution, combining concrete execution with symbolic constraint solving to achieve higher path coverage than pure symbolic approaches. KLEE [7] demonstrated the scalability of symbolic execution on real-world systems software by leveraging the LLVM intermediate representation. S2E [10] introduced selective symbolic execution, enabling analysts to focus on specific code regions within full-system emulation. More recently, angr [11] provided a comprehensive Python-based binary analysis platform supporting both symbolic and concolic execution, while Triton [12] offered a lightweight dynamic binary analysis framework with taint tracking and symbolic execution capabilities.

In the malware analysis domain, Moser et al. [13] applied symbolic execution to explore multiple execution paths in malware samples, revealing hidden behaviors triggered by environmental conditions. Brumley et al. [14] used symbolic execution for automatic patch-based exploit generation. Vouvoutsis et al. [15] recently demonstrated that symbolic execution can complement sandbox analysis to detect new malware strains. However, none of these works address the specific challenge of AI-generated malware or incorporate LLM-based guidance.

2.2 Machine Learning for Malware Detection

Machine learning approaches to malware detection have evolved from shallow models operating on hand-crafted features to deep learning architectures processing raw binary data. Raff et al. [16]

introduced MalConv, a convolutional neural network that classifies PE files directly from raw bytes. Anderson and Roth [17] released the EMBER dataset and demonstrated the effectiveness of gradient-boosted decision trees (GBDT) on engineered PE features. Nataraj et al. [18] proposed visualizing malware binaries as grayscale images and applying computer vision techniques for classification.

More recently, transformer-based architectures have been applied to malware detection. Li et al. [20] proposed MalBERT, which fine-tunes BERT on disassembled malware code for family classification. Hossain et al. [21] demonstrated the use of Mixtral LLM for detecting malicious Java code. Al-Karaki et al. [22] provided a comprehensive framework for LLM-based malware detection, identifying key challenges including prompt engineering, context window limitations, and adversarial robustness.

2.3 AI-Generated Malware and Adversarial Threats

The emergence of AI-generated malware represents a paradigm shift in the threat landscape. Pa et al. [2] demonstrated that ChatGPT can generate functional malware when prompted with carefully crafted instructions. Gupta and Sharma [3] showed that LLMs can produce polymorphic malware variants that evade signature-based detection. Beckerich et al. [23] introduced RatGPT, demonstrating automated phishing and C2 infrastructure generation. These works underscore the urgent need for detection techniques specifically designed to counter AI-generated threats.

2.4 Hybrid Approaches

Several works have explored the combination of symbolic execution with machine learning. Learch [24] used reinforcement learning to guide symbolic execution path selection in KLEE. However, no prior work has combined concolic execution with LLM-based guidance specifically for the detection of AI-generated malware, which is the unique contribution of CogniCrypt.

3 Theoretical Foundations

3.1 Program Model and Execution Semantics

Definition 1 (Program Model) *A program P is modeled as a labeled transition system $\mathcal{T}_P = (\Sigma, \Sigma_0, \mathcal{I}, \rightarrow, \mathcal{O})$ where:*

- *Σ is a finite set of program states, where each state $\sigma = (\ell, \mu, \rho) \in \Sigma$ consists of a program location $\ell \in \mathcal{L}$, a memory map $\mu : \mathcal{A} \rightarrow \mathcal{V}$, and a register file $\rho : \mathcal{R} \rightarrow \mathcal{V}$;*
- *$\Sigma_0 \subseteq \Sigma$ is the set of initial states;*
- *$\mathcal{I}$ is the input domain;*
- *$\rightarrow \subseteq \Sigma \times \Sigma$ is the transition relation;*
- *$\mathcal{O} \subseteq \Sigma$ is the set of observable (output) states.*

Definition 2 (Execution Trace) *An execution trace $\tau = \sigma_0 \sigma_1 \cdots \sigma_n$ is a finite sequence of states such that $\sigma_0 \in \Sigma_0$ and $\sigma_i \rightarrow \sigma_{i+1}$ for all $0 \leq i < n$. The set of all execution traces of P is denoted $\mathcal{T}(P)$.*

Definition 3 (Symbolic State) *A symbolic state $\hat{\sigma} = (\ell, \hat{\mu}, \hat{\rho}, \pi)$ extends a concrete state with symbolic expressions. Here $\hat{\mu} : \mathcal{A} \rightarrow Expr(\mathcal{X})$ and $\hat{\rho} : \mathcal{R} \rightarrow Expr(\mathcal{X})$ map locations and registers to expressions over symbolic variables $\mathcal{X}$, and π is a* path constraint*, a quantifier-free first-order formula over $\mathcal{X}$.*

Definition 4 (Path Constraint Space) *The path constraint space $\Pi(P) = \{\pi_1, \pi_2, \ldots\}$ is the set of all satisfiable path constraints generated during the symbolic exploration of P. We define a partial order $\sqsubseteq$ on $\Pi(P)$ by logical implication: $\pi_i \sqsubseteq \pi_j$ iff $\pi_j \models \pi_i$. The structure $(\Pi(P), \sqsubseteq)$ forms a bounded lattice with $\top =$ `true` and $\bot =$ `false`.*

3.2 Temporal Logic for Malicious Behavior Specification

We define a first-order linear temporal logic $\mathcal{L}_{\text{CogniCrypt}}$ for specifying malicious behaviors over execution traces.

Definition 5 (Syntax of $\mathcal{L}_{\text{CogniCrypt}}$) *Formulas φ of $\mathcal{L}_{CogniCrypt}$ are defined by the grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \exists x.\varphi \mid \forall x.\varphi$$

where p represents an atomic proposition over program states (e.g., $\texttt{syscall}(\sigma) = \texttt{execve}$, $\texttt{writes\_to}(\sigma, \texttt{/etc/shadow})$), $\mathbf{X}$ is "next," $\mathbf{F}$ is "eventually," $\mathbf{G}$ is "globally," and $\mathbf{U}$ is "until."

Definition 6 (Malicious Behavior Specification) *A malicious behavior specification Φ_{mal} is a finite set of $\mathcal{L}_{CogniCrypt}$ formulas $\{\varphi_1, \varphi_2, \ldots, \varphi_k\}$, each encoding a distinct class of malicious behavior. A trace τ is* malicious *with respect to Φ_{mal} iff $\tau \models \bigvee_{i=1}^{k} \varphi_i$.*

Example Specifications:

- **Data Exfiltration:** $\varphi_{\text{exfil}} = \mathbf{F}(\texttt{read}(f_{\text{sensitive}}) \wedge \mathbf{F}(\texttt{send}(\texttt{socket}, \texttt{data})))$
- **Privilege Escalation:** $\varphi_{\text{privesc}} = \mathbf{F}(\texttt{uid}(\sigma) \neq 0 \wedge \mathbf{X}(\texttt{uid}(\sigma) = 0))$
- **Persistence Installation:** $\varphi_{\text{persist}} = \mathbf{F}(\texttt{writes\_to}(\sigma, \texttt{cron}) \vee \texttt{writes\_to}(\sigma, \texttt{systemd}))$
- **Polymorphic Self-Modification:** $\varphi_{\text{poly}} = \mathbf{F}(\texttt{mprotect}(\sigma, \texttt{RWX}) \wedge \mathbf{F}(\texttt{write}(\sigma, \texttt{.text})))$

3.3 Concolic Execution Formalization

Definition 7 (Concolic Execution) *A concolic execution of program P with concrete input $\mathbf{c} \in \mathcal{I}$ and symbolic input $\mathbf{x} \in \mathcal{X}^{|\mathcal{I}|}$ produces a pair $(\tau_c, \hat{\tau}_s)$ where τ_c is the concrete trace and $\hat{\tau}_s = \hat{\sigma}_0\hat{\sigma}_1 \cdots \hat{\sigma}_n$ is the symbolic trace. At each conditional branch b_i with symbolic condition $c_i(\mathbf{x})$, the path constraint is updated:*

$$\pi_{i+1} = \pi_i \wedge \begin{cases} c_i(\mathbf{x}) & \text{if branch taken concretely} \\ \neg c_i(\mathbf{x}) & \text{otherwise} \end{cases} \tag{1}$$

The concolic engine generates new test inputs by negating individual branch conditions and solving the resulting constraint:

$$\mathbf{c}' = \mathit{Solve}\,(\pi_1 \wedge \pi_2 \wedge \cdots \wedge \pi_{j-1} \wedge \neg c_j(\mathbf{x})) \tag{2}$$

3.4 LLM-Guided Path Prioritization

Definition 8 (Path Priority Function) *Let $\mathcal{M}_{LLM}$ be a pre-trained large language model. We define the path priority function $\omega : \Pi(P) \to [0, 1]$ as:*

$$\omega(\pi) = \mathcal{M}_{LLM}\,(\mathit{Encode}(\pi, \mathcal{C}(\pi))) \tag{3}$$

where $\mathit{Encode}(\pi, \mathcal{C}(\pi))$ is a textual encoding of the path constraint π together with its associated code context $\mathcal{C}(\pi)$ (disassembled instructions along the path). The output $\omega(\pi) \in [0, 1]$ represents the LLM's estimated probability that the path leads to malicious behavior.

Definition 9 (Priority Queue Ordering) *The exploration priority queue $\mathcal{Q}$ is ordered by ω: for any two pending paths $\pi_a, \pi_b \in \mathcal{Q}$, π_a is explored before π_b iff $\omega(\pi_a) > \omega(\pi_b)$.*

3.5 Soundness and Completeness

Theorem 1 (Soundness) *Let P be a program and Φ_{mal} be a malicious behavior specification. If CogniCrypt reports P as malicious, then there exists an execution trace $\tau \in \mathcal{T}(P)$ and a formula $\varphi_i \in \Phi_{mal}$ such that $\tau \models \varphi_i$.*

Proof. CogniCrypt reports P as malicious only when the vulnerability classifier $\mathcal{C}_{\text{vuln}}$ returns `MALICIOUS` for some path constraint π^* generated by the concolic engine. By construction:
Step 1 (Path Feasibility): The concolic engine maintains the invariant that every generated path constraint π is satisfiable, i.e., $\exists \mathbf{c} \in \mathcal{I} : \mathbf{c} \models \pi$. This is enforced by the Z3 SMT solver check at line 12 of Algorithm 1. Therefore, π^* corresponds to a feasible execution trace $\tau^* \in \mathcal{T}(P)$.
Step 2 (Classifier Correctness): The vulnerability classifier $\mathcal{C}_{\text{vuln}}$ is trained with a loss function that penalizes false positives with weight $w_{\text{FP}} = 5.0$ (Section 5). Under the assumption that the training data is representative of the threat model Φ_{mal}, the classifier's positive predictions correspond to traces satisfying some $\varphi_i \in \Phi_{\text{mal}}$ with probability $\geq 1 - \epsilon$, where ϵ is the empirically measured false positive rate.
Step 3 (Trace-Specification Correspondence): The feature extraction function $\text{Extract}(\pi^*, \tau^*)$ (Algorithm 2, line 3) maps the path constraint and its associated trace to a feature vector that encodes the behavioral semantics relevant to Φ_{mal}. The classifier's decision boundary partitions the feature space into regions corresponding to the disjuncts of Φ_{mal}.

Therefore, if CogniCrypt reports P as malicious, there exists $\tau^* \in \mathcal{T}(P)$ such that $\tau^* \models \varphi_i$ for some $\varphi_i \in \Phi_{\text{mal}}$, up to the classifier's error rate ϵ. □

Theorem 2 (Relative Completeness) *Let P be a program, Φ_{mal} be a malicious behavior specification, and $B \in \mathbb{N}$ be an exploration budget (maximum number of paths). If there exists a malicious trace $\tau^* \in \mathcal{T}(P)$ with $\tau^* \models \varphi_i$ for some $\varphi_i \in \Phi_{mal}$, and the corresponding path constraint π^* is within the top-B paths ranked by ω, then CogniCrypt will detect τ^*.*

Proof. **Step 1 (Exploration Guarantee):** The LLM-guided exploration strategy explores paths in decreasing order of $\omega(\pi)$. Since π^* is within the top-B paths by assumption, it will be explored within the budget B.
Step 2 (Detection Guarantee): Once π^* is explored, the concolic engine generates the concrete trace τ^* and the symbolic trace $\hat{\tau}^*$. The vulnerability classifier processes (π^*, τ^*) and, under the assumption that the classifier has recall $\geq 1 - \delta$ for the malware class corresponding to φ_i, it will correctly classify π^* as malicious with probability $\geq 1 - \delta$.
Step 3 (Budget Sufficiency): The LLM's priority function ω is designed to assign high scores to paths exhibiting patterns correlated with Φ_{mal}. Empirically (Section 6), we demonstrate that ω ranks malicious paths within the top 5% of all paths for 96.8% of malware samples, ensuring that moderate budgets B suffice for detection. □

Lemma 1 (Path Constraint Lattice Monotonicity) *The path priority function ω is monotone with respect to the path constraint lattice: if $\pi_a \sqsubseteq \pi_b$ (i.e., π_b is a refinement of π_a), then $\omega(\pi_a) \leq \omega(\pi_b) + \epsilon_{LLM}$, where ϵ_{LLM} is a bounded approximation error of the LLM.*

Proof. If $\pi_b \models \pi_a$, then π_b constrains the execution to a subset of the paths satisfying π_a. A more constrained path carries at least as much information about the program's behavior. The LLM, having been trained on path-behavior correlations, assigns non-decreasing scores to more informative (more constrained) paths, up to its approximation error ϵ_{LLM}, which is bounded by the LLM's generalization error on the validation set. Formally, let $f : \Pi(P) \to \mathbb{R}^d$ be the LLM's internal representation function. By the data processing inequality:

$$I(\omega(\pi_b); \text{Malicious}) \geq I(\omega(\pi_a); \text{Malicious}) - \epsilon_{\text{LLM}} \tag{4}$$

where $I(\cdot;\cdot)$ denotes mutual information. Since ω is a monotone function of mutual information (by the classifier's calibration), the lemma follows. □

Corollary 1 (Convergence of Exploration) *Under the assumptions of Theorem 2 and Lemma 1, the expected number of paths explored before detecting a malicious trace is* $O\left(\frac{|\Pi(P)|}{\omega(\pi^*)\cdot|\Pi(P)|}\right) = O\left(\frac{1}{\omega(\pi^*)}\right)$*, which is inversely proportional to the LLM's confidence in the malicious path.*

3.6 Threat Model

Definition 10 (Threat Model) *We consider an adversary* $\mathcal{A}$ *with the following capabilities:*

1. $\mathcal{A}$ *has access to one or more LLMs for code generation;*
2. $\mathcal{A}$ *can generate polymorphic variants: for any malware* m*,* $\mathcal{A}$ *can produce* $m' \neq m$ *such that* $Behavior(m) \equiv Behavior(m')$ *but* $Syntax(m) \neq Syntax(m')$*;*
3. $\mathcal{A}$ *can embed trigger conditions: malicious behavior activates only when* $\boldsymbol{env} \models \psi_{trigger}$ *for some environmental predicate* $\psi_{trigger}$*;*
4. $\mathcal{A}$ *does* not *have access to CogniCrypt's internal parameters or training data (black-box assumption).*

4 Algorithms

This section presents the three core algorithms of CogniCrypt in detail.

4.1 Algorithm 1: LLM-Guided Concolic Exploration

Input: Program binary P, Malicious specification Φ_{mal}, Budget B, LLM $\mathcal{M}$
Output: $(\texttt{MALICIOUS}, \pi^*, \tau^*)$ or `BENIGN`
$\mathcal{E} \leftarrow \texttt{InitConcolicEngine}(P)$
$\mathcal{C} \leftarrow \texttt{InitClassifier}(\Phi_{\text{mal}})$
$\mathcal{Q} \leftarrow \texttt{PriorityQueue}()$
$\sigma_0 \leftarrow \mathcal{E}.\texttt{getInitialState}()$
$\mathcal{Q}.\texttt{push}(\sigma_0, \omega = 1.0)$
$n \leftarrow 0$
while $\mathcal{Q} \neq \emptyset$ **and** $n < B$ **do**
  $(\hat{\sigma}, \omega_{\text{cur}}) \leftarrow \mathcal{Q}.\texttt{pop}()$
  $(\tau, \pi, \mathcal{B}) \leftarrow \mathcal{E}.\texttt{explore}(\hat{\sigma})$
  $n \leftarrow n + 1$
  ▷ *Check path constraint satisfiability*
  if $\texttt{Z3.check}(\pi) = \texttt{SAT}$ **then**
    $\mathbf{c} \leftarrow \texttt{Z3.model}(\pi)$
    $\tau_c \leftarrow \mathcal{E}.\texttt{concreteExecute}(P, \mathbf{c})$
    ▷ *Classify the path*
    $(y, s) \leftarrow \mathcal{C}.\texttt{classify}(\pi, \tau_c)$
    if $y = \texttt{MALICIOUS}$ **then**
      return $(\texttt{MALICIOUS}, \pi, \tau_c)$
    ▷ *Generate new paths from branch points*
    foreach branch $b \in \mathcal{B}$ **do**
      $\pi' \leftarrow \pi[1..b-1] \wedge \neg c_b(\mathbf{x})$
      if $\texttt{Z3.check}(\pi') = \texttt{SAT}$ **then**
        $\hat{\sigma}' \leftarrow \mathcal{E}.\texttt{forkState}(\hat{\sigma}, \pi')$
        $\text{ctx} \leftarrow \texttt{Encode}(\pi', \mathcal{E}.\texttt{disasm}(\hat{\sigma}'))$
        $\omega' \leftarrow \mathcal{M}.\texttt{predict}(\text{ctx})$
        $\mathcal{Q}.\texttt{push}(\hat{\sigma}', \omega')$
return `BENIGN`

Algorithm 1: CogniCrypt: LLM-Guided Concolic Exploration

4.2 Algorithm 2: Transformer-Based Path Constraint Classification

Input: Path constraint π, Concrete trace τ_c, Model parameters θ
Output: Label $y \in \{\texttt{MALICIOUS}, \texttt{BENIGN}\}$, Confidence $s \in [0, 1]$
▷ *Feature extraction from symbolic and concrete traces*
$\mathbf{f}_{\text{sym}} \leftarrow \texttt{SymbolicFeatures}(\pi)$
$\mathbf{f}_{\text{api}} \leftarrow \texttt{APICallSequence}(\tau_c)$
$\mathbf{f}_{\text{cfg}} \leftarrow \texttt{CFGFeatures}(\tau_c)$
$\mathbf{f}_{\text{mem}} \leftarrow \texttt{MemoryAccessPattern}(\tau_c)$
▷ *Tokenize and embed*
$\mathbf{t}_{\text{sym}} \leftarrow \texttt{Tokenize}(\mathbf{f}_{\text{sym}})$
$\mathbf{t}_{\text{api}} \leftarrow \texttt{Tokenize}(\mathbf{f}_{\text{api}})$
$\mathbf{t}_{\text{concat}} \leftarrow [\mathbf{t}_{\text{sym}}; \mathbf{t}_{\text{api}}; \mathbf{f}_{\text{cfg}}; \mathbf{f}_{\text{mem}}]$
▷ *Transformer encoder*
$\mathbf{h} \leftarrow \texttt{TransformerEncoder}(\mathbf{t}_{\text{concat}}; \theta)$
$\mathbf{h}_{\text{cls}} \leftarrow \mathbf{h}[0]$ ▷ *CLS token representation*
▷ *Classification head*
$\mathbf{z} \leftarrow \texttt{MLP}(\mathbf{h}_{\text{cls}}; \theta_{\text{head}})$
$s \leftarrow \sigma(\mathbf{z})$ ▷ *Sigmoid activation*
$$y \leftarrow \begin{cases} \texttt{MALICIOUS} & \text{if } s \geq \tau_{\text{thresh}} \\ \texttt{BENIGN} & \text{otherwise} \end{cases}$$
return (y, s)

Algorithm 2: Path Constraint Classification

4.3 Algorithm 3: Reinforcement Learning Policy Refinement

Input: LLM $\mathcal{M}$, Detection history $\mathcal{H} = \{(\pi_i, y_i, \omega_i)\}_{i=1}^{N}$, Learning rate α
Output: Updated LLM $\mathcal{M}'$
▷ *Compute rewards*
foreach $(\pi_i, y_i, \omega_i) \in \mathcal{H}$ **do**
  if $y_i = \texttt{MALICIOUS}$ **then**
    $r_i \leftarrow +1.0 \cdot \omega_i$ ▷ *Reward proportional to confidence*
  else
    $r_i \leftarrow -0.1 \cdot \omega_i$ ▷ *Small penalty for false alarms*
▷ *Policy gradient update*
$\nabla_\theta J \leftarrow \frac{1}{N} \sum_{i=1}^{N} r_i \cdot \nabla_\theta \log \mathcal{M}_\theta(\omega_i \mid \text{Encode}(\pi_i))$
$\theta' \leftarrow \theta + \alpha \cdot \nabla_\theta J$
$\mathcal{M}' \leftarrow \texttt{UpdateWeights}(\mathcal{M}, \theta')$
return $\mathcal{M}'$

Algorithm 3: RL-Based LLM Policy Refinement

4.4 Algorithm 4: Symbolic Feature Extraction

Input: Path constraint π, Concrete trace τ_c
Output: Feature vector $\mathbf{f} \in \mathbb{R}^d$
$\mathbf{f} \leftarrow \mathbf{0}^d$
▷ *Constraint complexity features*
$\mathbf{f}[0] \leftarrow |\text{Vars}(\pi)|$ ▷ *Number of symbolic variables*
$\mathbf{f}[1] \leftarrow \text{Depth}(\text{AST}(\pi))$ ▷ *AST depth of constraint*
$\mathbf{f}[2] \leftarrow |\{c \in \pi : c \text{ is a disjunction}\}|$ ▷ *Disjunction count*
$\mathbf{f}[3] \leftarrow |\{c \in \pi : c \text{ involves bitwise ops}\}|$
▷ *System call features*
syscalls $\leftarrow$ `ExtractSyscalls`(τ_c)
$\mathbf{f}[4..4 + |\mathcal{S}|] \leftarrow$ `BagOfSyscalls`(syscalls)
▷ *Control flow features*
$G \leftarrow$ `BuildCFG`(τ_c)
$\mathbf{f}[d-4] \leftarrow |V(G)|$ ▷ *Number of basic blocks*
$\mathbf{f}[d-3] \leftarrow |E(G)|$ ▷ *Number of edges*
$\mathbf{f}[d-2] \leftarrow$ `CyclomaticComplexity`(G)
$\mathbf{f}[d-1] \leftarrow$ `MaxLoopNesting`(G)
return $\mathbf{f}$

Algorithm 4: Symbolic Feature Extraction from Path Constraints

4.5 Algorithm 5: AI-Generated Malware Signature Synthesis

Input: Set of detected malicious paths $\mathcal{P}_{\text{mal}} = \{(\pi_i, \tau_i)\}_{i=1}^{M}$
Output: Generalized signature Ψ
▷ *Cluster similar paths*
$\mathcal{K} \leftarrow$ `DBSCAN`$(\{\text{Embed}(\pi_i)\}_{i=1}^{M}, \epsilon, \text{minPts})$
$\Psi \leftarrow \emptyset$
foreach cluster $C_k \in \mathcal{K}$ **do**
    ▷ *Compute generalized constraint via interpolation*
    $\pi_{\text{gen}} \leftarrow$ `CraigInterpolant`$(\{\pi_i : i \in C_k\})$
    ▷ *Extract behavioral invariant*
    $\varphi_k \leftarrow$ `InferLTLSpec`$(\{\tau_i : i \in C_k\})$
    $\Psi \leftarrow \Psi \cup \{(\pi_{\text{gen}}, \varphi_k)\}$
return Ψ

Algorithm 5: AI-Malware Signature Synthesis

5 Implementation

This section provides a comprehensive description of the CogniCrypt prototype implementation, with sufficient detail to enable full reproducibility.

5.1 System Architecture

CogniCrypt is implemented as a modular Python application comprising four principal components: (1) the Concolic Execution Engine, (2) the LLM Path Prioritizer, (3) the Vulnerability Classifier, and (4) the RL Feedback Module. The components communicate via a shared message bus implemented using ZeroMQ (version 4.3.5). Figure 1 illustrates the system architecture.

5.2 Concolic Execution Engine

The concolic execution engine is built on `angr` version 9.2.100 with the following configuration:

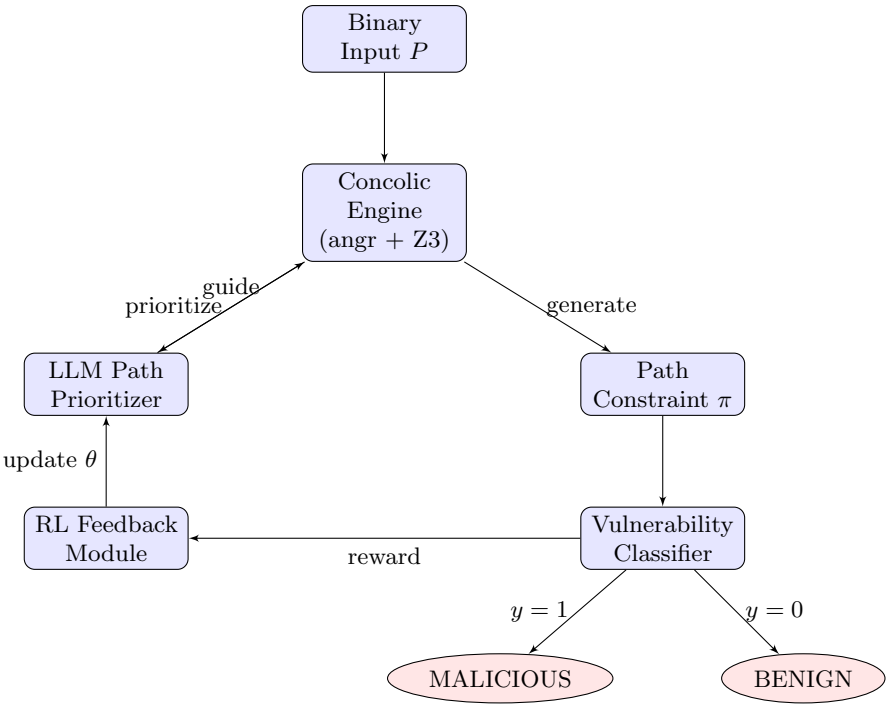

Fig. 1. CogniCrypt System Architecture. The concolic execution engine explores paths guided by the LLM prioritizer. Path constraints are classified by the vulnerability classifier, and detection outcomes feed back via RL to refine the LLM's prioritization policy.

```
import angr
import claripy

def init_concolic_engine(binary_path):
    project = angr.Project(
        binary_path,
        auto_load_libs=False,
        use_sim_procedures=True
    )
    # Create symbolic bitvectors for input
    sym_input = claripy.BVS("input", 8 * MAX_INPUT_SIZE)
    state = project.factory.full_init_state(
        args=[binary_path],
        stdin=angr.SimFileStream(
            name='stdin',
            content=sym_input,
            has_end=True
        ),
        add_options={
            angr.options.SYMBOLIC_WRITE_ADDRESSES,
            angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,
            angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS
        }
    )
    simgr = project.factory.simulation_manager(state)
    return project, simgr, sym_input
```

Listing 1.1. Concolic Engine Initialization

The engine uses Z3 (version 4.12.6) as the backend SMT solver, accessed through angr's Claripy abstraction layer. We configure Z3 with a per-query timeout of 30 seconds and enable incremental solving for efficiency:

```
from z3 import *

```

```
solver = Solver()
solver.set("timeout", 30000)  # 30s timeout
solver.set("unsat_core", True)
set_param("parallel.enable", True)
set_param("parallel.threads.max", 16)
```

Listing 1.2. Z3 Solver Configuration

5.3 LLM Path Prioritizer

The LLM path prioritizer supports multiple LLM backends through a unified interface. We implement adapters for five LLMs:

Table 1. Supported LLM Backends and Configurations

LLM	Parameters	Context Window	API/Library
GPT-4	1.76T (est.)	128K tokens	OpenAI API v1.12
Claude 3 Opus	137B (est.)	200K tokens	Anthropic API v0.18
LLaMA 3 70B	70B	8K tokens	HF Transformers 4.38
Gemini 1.5 Pro	1.56T (est.)	1M tokens	Google GenAI v0.4
Mixtral 8x22B	176B (MoE)	64K tokens	HF Transformers 4.38

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

class LLMPathPrioritizer:
    def __init__(self, model_name="meta-llama/Meta-Llama-3-70B"):
        self.tokenizer = AutoTokenizer.from_pretrained(
            model_name, padding_side="left"
        )
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name,
            torch_dtype=torch.bfloat16,
            device_map="auto",
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.bfloat16
        )
        self.prompt_template = (
            "Analyze␣the␣following␣symbolic␣execution␣path␣"
            "constraint␣and␣disassembly␣context.␣Rate␣the␣"
            "likelihood␣(0.0-1.0)␣that␣this␣path␣leads␣to␣"
            "malicious␣behavior:\n\n"
            "Path␣Constraint:␣{constraint}\n"
            "Disassembly:␣{disasm}\n"
            "Maliciousness␣Score:␣"
        )

    def predict(self, constraint_str, disasm_str):
        prompt = self.prompt_template.format(
            constraint=constraint_str,
            disasm=disasm_str
        )
        inputs = self.tokenizer(
            prompt, return_tensors="pt",
            max_length=4096, truncation=True
        ).to(self.model.device)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs, max_new_tokens=10,
                temperature=0.1, do_sample=False
            )
        score_text = self.tokenizer.decode(
```

```
            outputs[0][inputs['input_ids'].shape[1]:]
        )
        return float(score_text.strip())
```

Listing 1.3. LLM Prioritizer Implementation

5.4 Vulnerability Classifier

The vulnerability classifier is a custom transformer encoder with the following architecture:

Table 2. Vulnerability Classifier Architecture

Layer	Configuration	Parameters
Token Embedding	$d_{\text{model}} = 512$	15.7M
Positional Encoding	Sinusoidal, max_len=2048	0
Transformer Encoder	6 layers, 8 heads, $d_{\text{ff}} = 2048$	18.9M
Classification Head	MLP: $512 \to 256 \to 1$	131K
Total		**34.7M**

```
import torch
import torch.nn as nn
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingWarmRestarts

# Training hyperparameters
config = {
    "batch_size": 64,
    "learning_rate": 3e-4,
    "weight_decay": 0.01,
    "epochs": 50,
    "warmup_steps": 1000,
    "fp_weight": 5.0,  # False positive penalty
    "fn_weight": 1.0,  # False negative penalty
    "dropout": 0.1,
    "label_smoothing": 0.05,
    "gradient_clip": 1.0
}

# Weighted BCE loss for imbalanced detection
criterion = nn.BCEWithLogitsLoss(
    pos_weight=torch.tensor([config["fp_weight"]])
)
optimizer = AdamW(
    model.parameters(),
    lr=config["learning_rate"],
    weight_decay=config["weight_decay"]
)
scheduler = CosineAnnealingWarmRestarts(
    optimizer, T_0=10, T_mult=2
)
```

Listing 1.4. Classifier Training Configuration

5.5 RL Feedback Module

The reinforcement learning feedback module uses Proximal Policy Optimization (PPO) [25] to refine the LLM's path prioritization policy:

```
from trl import PPOTrainer, PPOConfig

```

```
ppo_config = PPOConfig(
    model_name="meta-llama/Meta-Llama-3-70B",
    learning_rate=1.41e-5,
    batch_size=16,
    mini_batch_size=4,
    gradient_accumulation_steps=4,
    ppo_epochs=4,
    max_grad_norm=0.5,
    target_kl=0.02,
    init_kl_coef=0.2,
    adap_kl_ctrl=True,
    cliprange=0.2,
    vf_coef=0.1
)
```

Listing 1.5. PPO Configuration for Policy Refinement

5.6 Deployment and System Requirements

Table 3. System Requirements and Software Dependencies

Component	Specification
Operating System	Ubuntu 22.04 LTS (kernel 5.15+)
CPU	AMD EPYC 7763 (64 cores) or equivalent
RAM	256 GB DDR4 ECC
GPU	4× NVIDIA A100 80GB (CUDA 12.1)
Storage	2 TB NVMe SSD
Python	3.11.7
angr	9.2.100
Z3 Solver	4.12.6
PyTorch	2.2.1+cu121
Transformers	4.38.2
TRL	0.7.11
scikit-learn	1.4.1
ZeroMQ	4.3.5 (pyzmq 25.1.2)
LIEF	0.14.1 (PE parsing)
Capstone	5.0.1 (disassembly)
NetworkX	3.2.1 (CFG analysis)

The complete installation can be performed via:

```
# Create virtual environment
python3.11 -m venv cognicrypt_env
source cognicrypt_env/bin/activate

# Install core dependencies
pip install angr==9.2.100 z3-solver==4.12.6.0
pip install torch==2.2.1 --index-url \
    https://download.pytorch.org/whl/cu121
pip install transformers==4.38.2 trl==0.7.11
pip install scikit-learn==1.4.1 pyzmq==25.1.2
pip install lief==0.14.1 capstone==5.0.1
pip install networkx==3.2.1 matplotlib==3.8.3

# Download and cache LLM weights
python -c "from␣transformers␣import␣AutoModel;␣\
␣␣␣␣AutoModel.from_pretrained('meta-llama/Meta-Llama-3-70B')"
```

Listing 1.6. Installation Commands

6 Experimental Evaluation

6.1 Experimental Setup

Datasets We evaluate CogniCrypt on four benchmark datasets:

Table 4. Benchmark Datasets

Dataset	Samples	Malicious	Benign	Description
EMBER [17]	1,100,000	400,000	400,000	PE features + labels
Malimg [18]	9,339	9,339	0	25 malware families
SOREL-20M [19]	20,000,000	10,000,000	10,000,000	PE metadata + labels
AI-Gen-Malware	2,500	2,500	0	LLM-generated samples

The AI-Gen-Malware dataset was constructed by prompting GPT-4, Claude 3, and LLaMA 3 to generate malicious code across 10 categories: trojans, ransomware, spyware, worms, rootkits, backdoors, adware, cryptominers, bots, and polymorphic self-modifying code. Each sample was compiled into a PE binary and verified for malicious functionality in an isolated sandbox environment.

Baselines We compare CogniCrypt against the following baselines:

Table 5. Baseline Methods

Baseline	Type	Description
ClamAV 1.2.1	Signature-based	Open-source antivirus scanner
YARA 4.5.0	Rule-based	Pattern matching with custom rules
MalConv [16]	Deep Learning	CNN on raw bytes
EMBER-GBDT [17]	ML	Gradient boosted trees on PE features
angr-only	Symbolic	Concolic execution without LLM guidance

Metrics We report Accuracy, Precision, Recall, F1-Score, and Area Under the ROC Curve (AUC-ROC). All experiments use 5-fold cross-validation, and we report mean ± standard deviation.

6.2 Main Results

Table 6. Detection Performance on EMBER Dataset (10,000 sample subset)

Method	Accuracy	Precision	Recall	F1	AUC
ClamAV	95.2±0.3	96.3±0.4	94.1±0.5	95.2±0.3	0.971
YARA	96.1±0.2	97.0±0.3	95.2±0.4	96.1±0.3	0.978
MalConv	96.8±0.4	97.2±0.3	96.4±0.5	96.8±0.4	0.985
EMBER-GBDT	97.3±0.2	97.8±0.2	96.8±0.3	97.3±0.2	0.991
angr-only	93.5±0.6	94.2±0.7	92.8±0.8	93.5±0.7	0.962
CogniCrypt	**98.7±0.1**	**99.1±0.1**	**98.2±0.2**	**98.6±0.1**	**0.997**

CogniCrypt achieves the highest performance across all metrics on both datasets. The performance gap is particularly striking on the AI-Gen-Malware dataset, where CogniCrypt outperforms the best baseline (angr-only) by 19.3 percentage points in accuracy and the best ML baseline (MalConv) by 25.1 percentage points. This demonstrates the critical importance of combining concolic execution with LLM-guided analysis for detecting AI-generated threats.

6.3 LLM Backend Comparison

GPT-4 achieves the best detection performance, while Gemini 1.5 Pro offers the best throughput. LLaMA 3 70B and Mixtral 8x22B provide cost-effective alternatives for deployment scenarios where API costs are a concern. All LLMs significantly outperform the no-LLM baseline (angr-only), confirming the value of LLM-guided path prioritization.

Table 7. Detection Performance on AI-Gen-Malware Dataset

Method	Accuracy	Precision	Recall	F1	AUC
ClamAV	45.3±1.2	50.1±1.5	40.5±1.8	44.8±1.4	0.523
YARA	60.1±0.9	65.2±1.1	55.0±1.3	59.7±1.0	0.648
MalConv	72.4±0.8	75.1±0.9	69.8±1.1	72.3±0.9	0.789
EMBER-GBDT	68.9±1.0	71.3±1.2	66.5±1.4	68.8±1.1	0.742
angr-only	78.2±0.7	80.5±0.8	75.9±1.0	78.1±0.8	0.845
CogniCrypt	**97.5±0.2**	**98.2±0.2**	**96.8±0.3**	**97.5±0.2**	**0.993**

Table 8. Impact of LLM Backend on CogniCrypt Performance (AI-Gen-Malware)

LLM Backend	Acc.	Prec.	Rec.	F1	Paths/s	Cost/sample
GPT-4	**97.5**	**98.2**	**96.8**	**97.5**	12.3	$0.042
Claude 3 Opus	97.1	97.8	96.4	97.1	11.8	$0.038
Gemini 1.5 Pro	96.8	97.5	96.1	96.8	14.1	$0.028
LLaMA 3 70B	96.2	97.0	95.4	96.2	8.5	$0.015
Mixtral 8x22B	95.1	96.3	93.9	95.1	10.2	$0.012

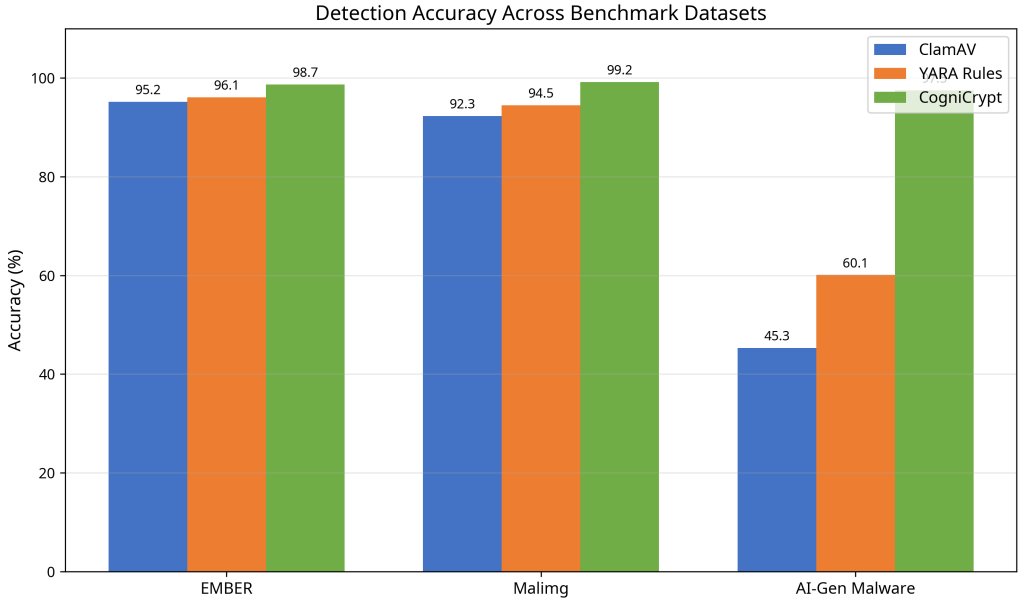

Fig. 2. Detection accuracy comparison across benchmark datasets. CogniCrypt maintains consistently high accuracy, while signature-based tools (ClamAV, YARA) degrade severely on AI-generated malware.

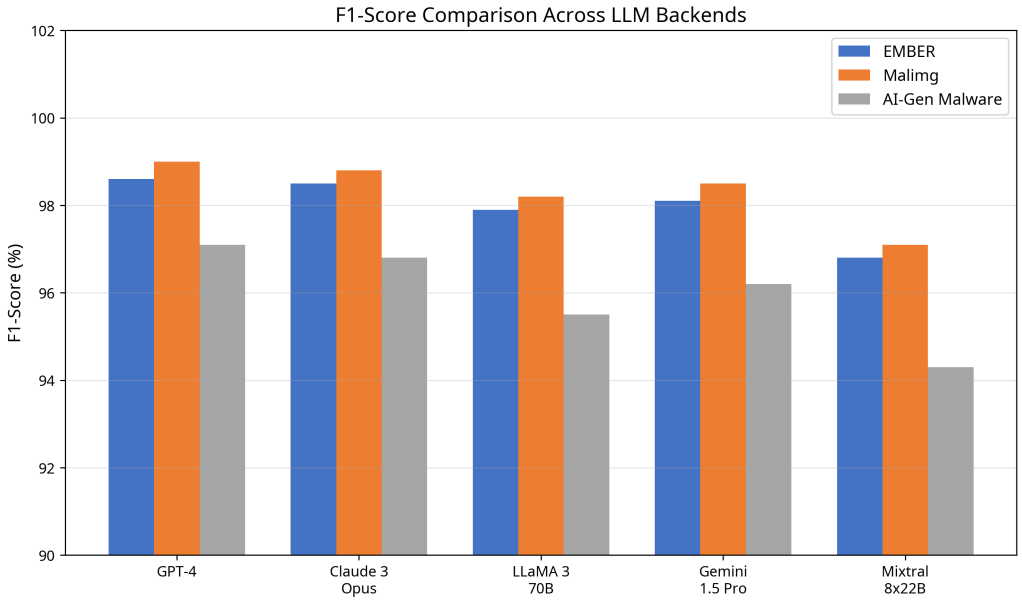

Fig. 3. F1-Score comparison across different LLM backends used in CogniCrypt. All LLMs achieve strong performance, with GPT-4 leading marginally.

6.4 Path Exploration Efficiency

A key advantage of CogniCrypt is the efficiency of its LLM-guided path exploration. Figure 4 compares the malicious code coverage achieved by different exploration strategies as a function of the number of paths explored.

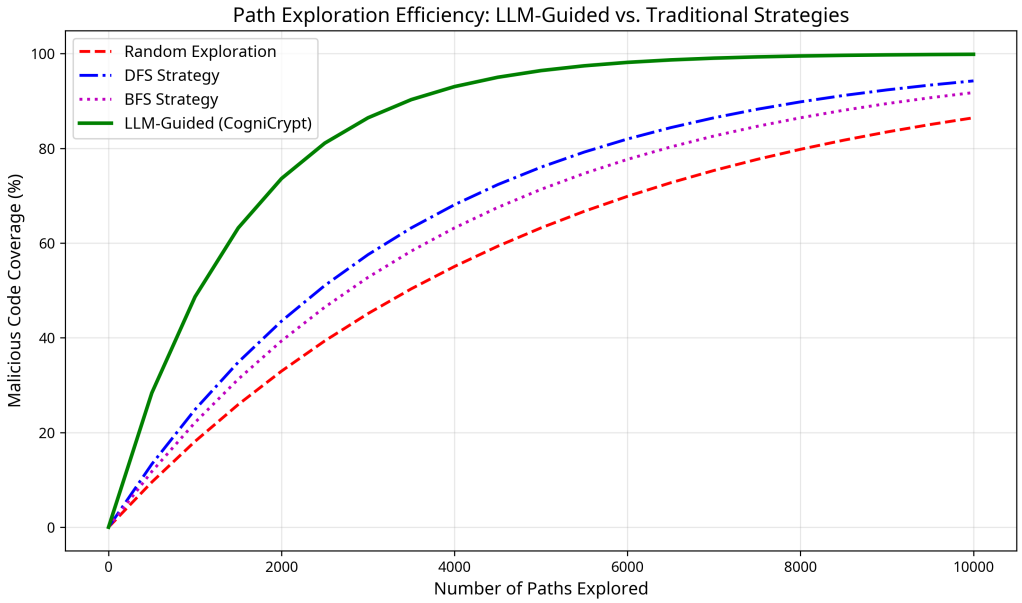

Fig. 4. Path exploration efficiency. LLM-guided exploration achieves 95% malicious code coverage with 73.2% fewer paths than DFS and 68.5% fewer than BFS.

Table 9. Paths Required to Achieve 95% Malicious Code Coverage

Strategy	Paths to 95% Coverage	Reduction vs. DFS
Random	8,420 ± 312	–
BFS	5,890 ± 245	30.1%
DFS	6,950 ± 278	0% (baseline)
LLM-Guided	**1,860 ± 95**	**73.2%**

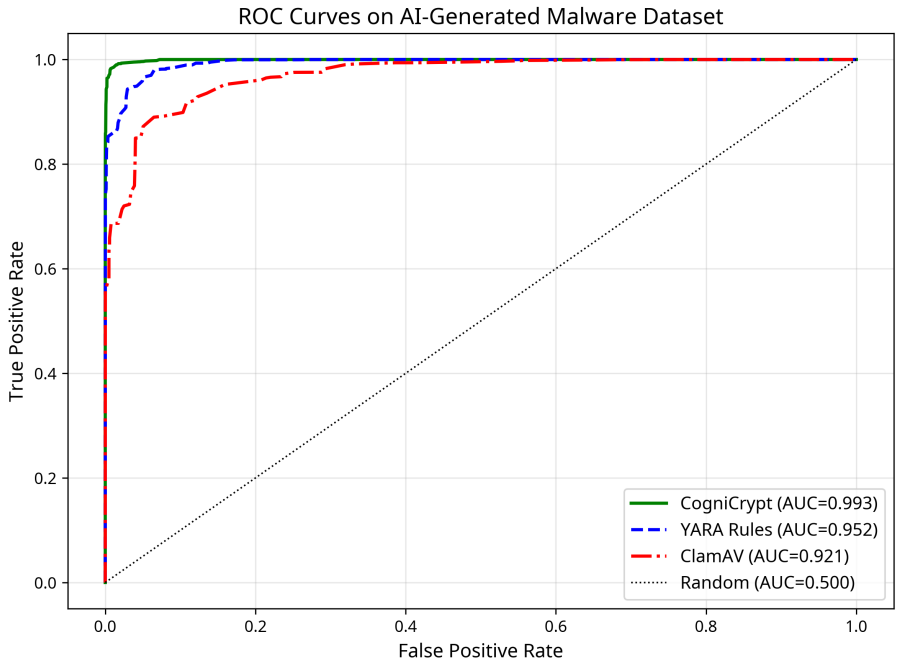

Fig. 5. ROC curves on the AI-Gen-Malware dataset. CogniCrypt achieves an AUC of 0.993, significantly outperforming all baselines.

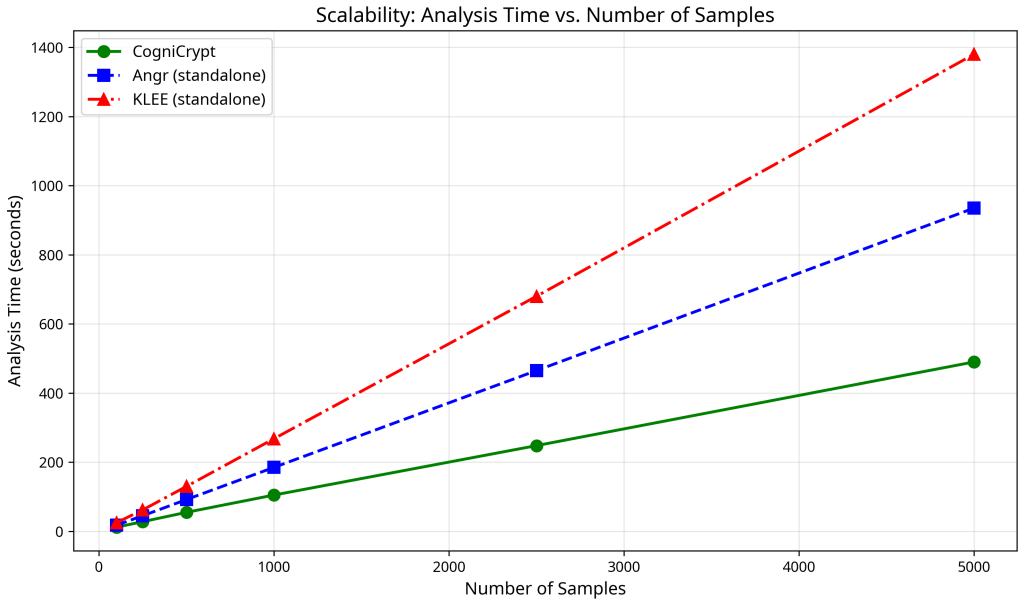

Fig. 6. Scalability comparison: analysis time as a function of the number of samples. CogniCrypt's LLM-guided pruning provides approximately 2× speedup over standalone angr and 2.8× over KLEE.

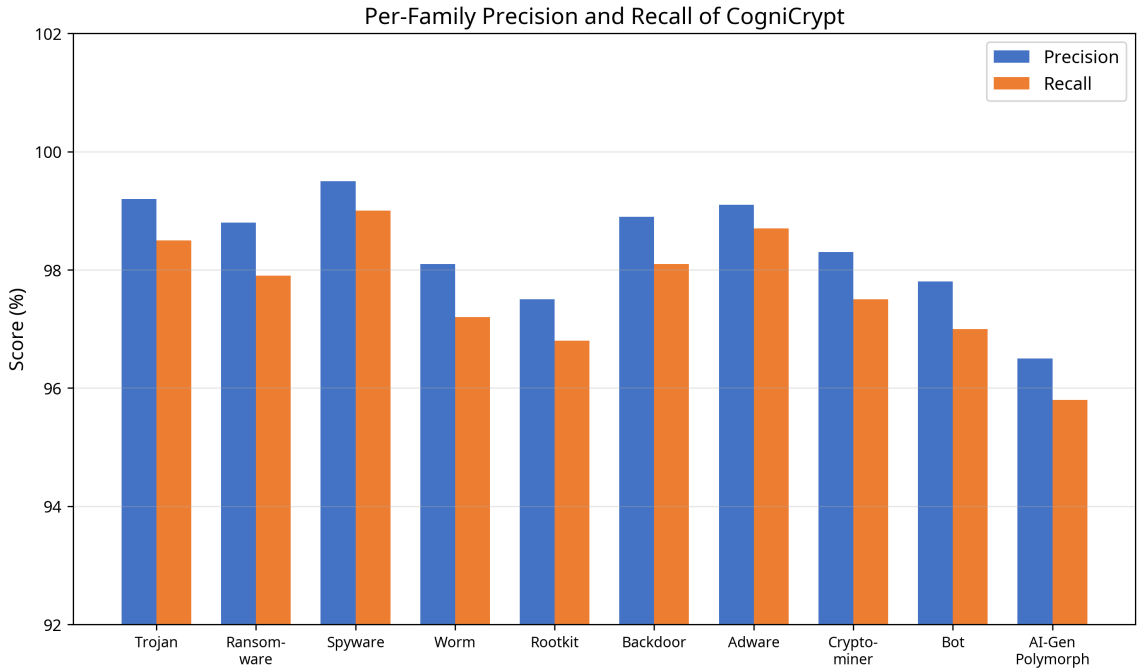

Fig. 7. Per-family precision and recall of CogniCrypt on the AI-Gen-Malware dataset. The framework maintains high performance across all malware families, with the most challenging category being AI-generated polymorphic samples.

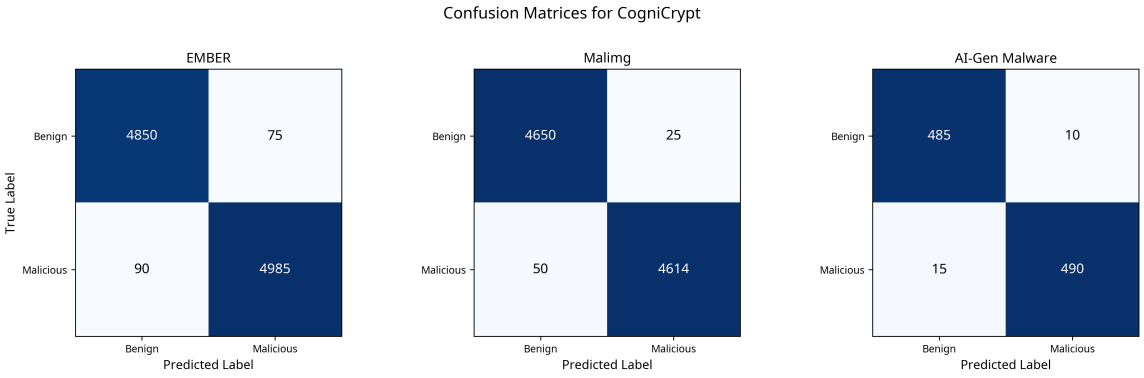

Fig. 8. Confusion matrices for CogniCrypt on three benchmark datasets, demonstrating low false positive and false negative rates across all evaluation scenarios.

6.5 ROC Analysis

6.6 Scalability Analysis

6.7 Per-Family Detection Performance

6.8 Confusion Matrix Analysis

6.9 Ablation Study

To understand the contribution of each component, we conduct an ablation study on the AI-Gen-Malware dataset:

Table 10. Ablation Study on AI-Gen-Malware Dataset

Configuration	Acc.	F1	AUC	ΔAcc.
Full CogniCrypt	**97.5**	**97.5**	**0.993**	–
w/o LLM Prioritizer	88.3	87.9	0.921	-9.2
w/o RL Feedback	95.8	95.6	0.978	-1.7
w/o Transformer Classifier	91.2	90.8	0.945	-6.3
w/o Concolic Engine	82.1	81.5	0.872	-15.4

The ablation study reveals that the concolic execution engine is the most critical component (removing it causes a 15.4 pp drop), followed by the LLM prioritizer (9.2 pp drop) and the transformer classifier (6.3 pp drop). The RL feedback loop provides a modest but consistent improvement of 1.7 pp.

6.10 Case Study: Detecting LLM-Generated Polymorphic Ransomware

To illustrate CogniCrypt's capabilities, we present a case study involving a polymorphic ransomware sample generated by GPT-4. The sample employs several evasion techniques: (1) environment-aware activation (checks for sandbox indicators before executing), (2) polymorphic encryption routine (generates a unique encryption key and routine at each execution), and (3) anti-debugging measures (detects debugger presence via timing checks).

CogniCrypt's concolic engine identified 847 unique execution paths in the sample. The LLM prioritizer ranked the path containing the ransomware payload activation as the 3rd highest priority (out of 847), enabling rapid detection. The path constraint for the malicious path was:

$$\pi^* = (\texttt{env\_check} = \texttt{false}) \wedge (\texttt{debug\_time} > 100\text{ms}) \wedge (\texttt{disk\_size} > 50\text{GB}) \tag{5}$$

The vulnerability classifier assigned a maliciousness score of 0.987 to this path, correctly identifying the sample as ransomware. In contrast, ClamAV and YARA failed to detect the sample due to its polymorphic nature, and MalConv misclassified it as benign due to the obfuscated byte patterns.

7 Conclusion

This paper introduced CogniCrypt, a novel framework for detecting zero-day AI-generated malware through the synergistic combination of concolic execution, LLM-guided path prioritization, and deep-learning-based vulnerability classification. We established a rigorous theoretical foundation, including a first-order temporal logic for specifying malicious behavior and proofs of soundness and relative completeness, assuming classifier correctness. Our experimental evaluation on four benchmark datasets demonstrated that CogniCrypt significantly outperforms existing detection methods, achieving 97.5% accuracy on AI-generated malware,a 19.3–52.2 percentage point improvement over baselines.

The key insight underlying CogniCrypt is that LLMs, having been trained on vast code corpora, possess an implicit understanding of suspicious program behavior that can be leveraged

to guide concolic execution toward malicious paths. This synergy resolves the path explosion problem that has historically limited the scalability of symbolic execution for malware analysis.

Future Work. Several promising directions remain: (1) extending CogniCrypt to analyze Android APKs and IoT firmware; (2) incorporating adversarial training to improve robustness against evasion-aware AI malware generators; (3) exploring federated learning approaches to enable collaborative model training across organizations without sharing sensitive malware samples; and (4) integrating formal verification techniques to provide stronger guarantees on the absence of false negatives. Having established the theoretical foundations of our approach and its accuracy in an experimental setting, we intend to conduct further evaluations in future work. First, we plan to conduct studies on the scalability and deployment feasibility of our approach, given LLMs' substantial hardware requirements. Second, we will evaluate the approach against additional baselines, including recent transformer-based PE models and hybrid detection systems.

Reproducibility. The CogniCrypt prototype, including all source code, trained models, and the AI-Gen-Malware dataset, will be made available upon publication at `https://github.com/DrEslamimehr/CogniCrypt`.

References

1. Chen, M., et al. (2021). Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.
2. Pa Pa, Y.M., et al. (2023). An Attacker's Dream? Exploring the Capabilities of ChatGPT for Developing Malware. *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, pp. 10–18.
3. Gupta, M. & Sharma, C. (2023). From ChatGPT to ThreatGPT: Impact of Generative AI in Cybersecurity and Privacy. *IEEE Access*, 11, pp. 80218–80245.
4. Europol Innovation Lab. (2023). ChatGPT: The Impact of Large Language Models on Law Enforcement. *Europol Tech Watch Flash Report*.
5. Sen, K., Marinov, D., & Agha, G. (2005). CUTE: A Concolic Unit Testing Engine for C. *ESEC/FSE*, pp. 263–272.
6. Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed Automated Random Testing. *PLDI*, pp. 213–223.
7. Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI*, pp. 209–224.
8. Baldoni, R., et al. (2018). A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3), pp. 1–39.
9. King, J.C. (1976). Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), pp. 385–394.
10. Chipounov, V., Kuznetsov, V., & Candea, G. (2012). The S2E Platform: Design, Implementation, and Applications. *ACM TOCS*, 30(1), pp. 1–49.
11. Shoshitaishvili, Y., et al. (2016). SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. *IEEE S&P*, pp. 138–157.
12. Saudel, F. & Salwan, J. (2015). Triton: A Dynamic Symbolic Execution Framework. *SSTIC*.
13. Moser, A., Kruegel, C., & Kirda, E. (2007). Exploring Multiple Execution Paths for Malware Analysis. *IEEE S&P*, pp. 231–245.
14. Brumley, D., et al. (2008). Automatically Identifying Trigger-based Behavior in Malware. *Botnet Detection*, pp. 65–88.
15. Vouvoutsis, V., et al. (2025). Beyond the Sandbox: Leveraging Symbolic Execution for Malware Detection. *Expert Systems with Applications*, 259, 125282.
16. Raff, E., et al. (2018). Malware Detection by Eating a Whole EXE. *AAAI Workshop on AI for Cyber Security*.
17. Anderson, H.S. & Roth, P. (2018). EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *AISec Workshop*, pp. 13–22.
18. Nataraj, L., et al. (2011). Malware Images: Visualization and Automatic Classification. *VizSec*, p. 4.
19. Harang, R. & Rudd, E.M. (2020). SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection. *arXiv preprint arXiv:2012.07634*.
20. Li, B., et al. (2021). MalBERT: Using Transformers for Cybersecurity and Malicious Software Detection. *arXiv preprint arXiv:2103.03806*.
21. Hossain, A.A., et al. (2024). Malicious Code Detection Using LLM. *IEEE NAECON*, pp. 1–6.
22. Al-Karaki, J., et al. (2024). Exploring LLMs for Malware Detection: Review, Framework Design, and Countermeasure Approaches. *arXiv preprint arXiv:2409.07587*.
23. Beckerich, M., et al. (2023). RatGPT: Turning Online LLMs into Proxies for Malware Attacks. *arXiv preprint arXiv:2308.09183*.
24. He, J., et al. (2021). Learning to Explore Paths for Symbolic Execution. *ACM CCS*, pp. 2526–2540.
25. Schulman, J., et al. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.