MPMalGen: A Framework for Multi-Platform LLM powered Malware Variant Generation

Md Mohaiminul Islam
mmislam@iastate.edu

Abstract

We extend LLM-based malware variant generation to multi-platform environments by introducing platform-specific transformation strategies for Windows and Android malware and enhanced processing and with a better task instructed LLM. Our framework achieves better AV evasion and acts as a proof of concept work for a unified framework for multi platform malware generation at mass scale.

1 Introduction

Malware evolves at a rapid pace, with adversaries constantly developing new variants to evade detection mechanisms. While antivirus engines and machine learning-based detectors form the backbone of modern defense strategies, attackers employ code obfuscation, polymorphism, and metamorphism to circumvent these protections. Recent advances in Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation and transformation tasks, raising concerns about their potential misuse in malware development. Malware authors with access to source code, whether through leaks, open-source repositories, or their own development - face fewer barriers to generating variants. In this work we want to address the problem: *"How do we alter or change malicious software source code in such a way that it has a better chance to evade antivirus shields and detectors while preserving its functional capacity"*. Specifically we want to use the code generation and small-scale refactoring ability of LLMs to improve the efficiency, semantic preservation, and evasion effectiveness of LLM-guided malware variant generation at the source-code level.

2 Related Work

Prior research on malware variant generation has explored multiple approaches with varying levels of automation and effectiveness. Qiao et al. (Qiao et al., 2022) proposed adversarial malware sample generation methods based on prototypes of deep learning detectors, while AMVG (Choi et al., 2019) introduced an adaptive malware variant generation framework using machine learning. Ming et al. (Ming et al., 2017) investigated impeding behavior-based malware analysis through replacement attacks on malware specifications. Binary-level transformation techniques such as Malware Makeover (Lucas et al., 2021) and MalGuise (Ling et al., 2024) employ adversarial machine learning and semantic transformation methods to modify malware binaries for evasion, though these approaches require extensive iterative optimization. Notably, Botacin et al. (Botacin, 2023) demonstrated that LLMs can generate malware code fragments from natural language prompts, but their approach suffers from low success rates and does not guarantee functional correctness.

3 Design and Implementation

3.1 Foundational Work: LLMalMorph

LLMalMorph(Akil et al., 2025) is a semi-automated framework that generates variant Windows malware at the source-code level by applying function-level transformations with a pre-trained large language model (LLM) and then recompiling the modified projects. The pipeline has two main modules: (1) the *Function Mutator*, which parses each malware file into an abstract syntax tree, extracts headers, global declarations, and individual function bodies, and then feeds each selected function plus contextual information to the LLM via carefully engineered prompts; and (2) the *Variant Synthesizer*, which incrementally merges LLM-modified functions back into the project, recompiles after each step, uses a human-in-the-loop debugging process to fix bugs without altering semantic meaning of the code.The authors evaluate LLMalMorph on 10 real-world Windows malware

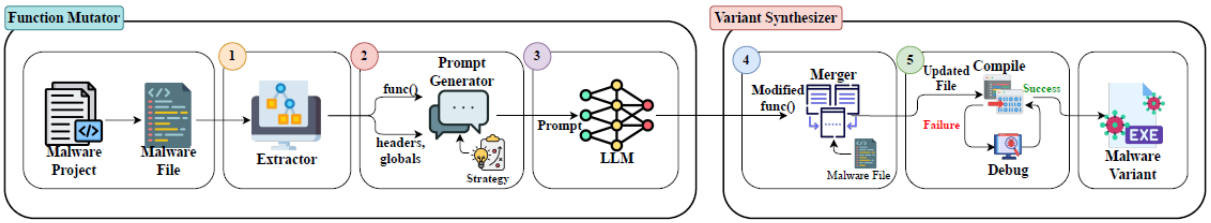

Figure 1: LLMalMorph framework

families with available *C/C++* source code generating 618 compiled variants in total.

We extend LLMalMorph's approach to address its limitations by introducing multi-platform support for Android environments, binary-level APK analysis capabilities, and integrating superior code generation models. Additionally, we propose automated rule-based and AI-assisted patching strategies that substantially reduce manual intervention. Our evaluation both replicates the original findings on Windows malware and expands the assessment to include newly collected Windows samples alongside comprehensive Android and APK threat analysis.

3.2 Our Proposal: MPMalGen Framework

We propose MPMALGEN (Multi-platform Malware Generation), which implements a multi-stage pipeline consisting of integrating source discovery code, structural parsing, LLM-based transformation, post-generation sanitization, and automated recompilation. The framework introduces a number of architectural contributions for an improved robust and reproducible generation of malware variants at scale.

Multi-platform Ingestion: Our framework implements a unified ingestion pipeline supporting diverse source code formats and intermediate code representations across multiple platforms. This cross-abstraction approach allows comparison of transformation strategies at both source and intermediate compilation levels, addressing whether semantic preservation is more reliably achieved at the source or intermediate stage. By abstracting platform-specific implementation details, the framework facilitates scalable processing of malware families independent of their target environment or code representation.

Android-Compatible Parsing Tree: We introduce a lightweight, schema-agnostic parser that extracts function signatures, parameter metadata, and class structure information without dependence on heavyweight grammar frameworks. This design dependss on the insight that full syntactic parsing is unnecessary for function-level transformation. Only signature extraction and boundary detection suffices to isolate transformable units. The parsed metadata is kept in structured format, to be reused as cache across experimental trials and reducing parser invocation overhead. For this module we mostly kept LLMalMorphs code, just following their lead to add android specific parsing improvements. MPMalGen can also parse assembly level smali codes.

Platform-Aware Prompt Engineering with Android-Specific Transformations: While prior work established a foundational context-aware prompting approach, we introduce a critical extension necessary for multi-platform compatibility which is platform-specific transformation strategy. The original five general-purpose strategies (Code Optimization, Quality Refinement, Modularization, Security, and Obfuscation) remain applicable across all platforms and are retained without modification. However, our primary contribution-the Android API Adaptation strategy recognizes that effective malware variant generation requires domain-specific constraints that reflect the unique architecture and execution model of Android environments. This novel strategy encodes Android-specific knowledge into the prompt template, influencing the variants. We added several key instructions such as: (1) preservation of component lifecycle semantics (Service/Receiver/Activity declarations and lifecycle callbacks), (2) maintenance of manifest registrations and method signatures, (3) safe transformation of permission models and intent-based communication patterns, (4) obfuscation of background execution mechanisms (JobScheduler, WorkManager, AlarmManager, foreground services) without disrupting functionality etc. and more to guide the LLM in generating func-

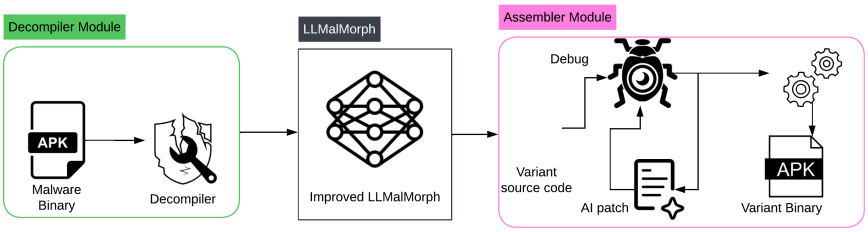

Figure 2: Improved MPMalGen Framework

tion level replacements, while maintaining compatibility with operational infrastructure. Injecting these Android-specific constraints into the prompt composition framework, we enable the LLM to generate semantically-correct variants that try to achieve evasion and remain functionally compatible with Android's strict component model and permission framework.

Selection of Expert Model: While prior work employed *Codestral-22B* for malware variant generation, we transition to *DeepSeek-Coder-v2-16B-Instruct*, an instruction-tuned model specifically optimized for code generation tasks. This selection is due to previous works demonstrating that DeepSeek-Coder-v2, despite its smaller 16B parameter footprint, demonstrates superior code comprehension and transformation/editing capabilities relative to Codestral-22B. Moreover, the model exhibits better proficiency in assembly-level code manipulation, a desired capability for handling intermediate bytecode representations. We configure the model with consistent hyperparameters across all experiments: temperature 0.8, top-k 40, and top-p 0.9, with a fixed random seed to ensure reproducibility. These settings are calibrated to balance generation diversity and allow exploration of transformation variants.

Automated Syntactic Validation and Error Recovery: The framework implements a multi-stage validation and repair pipeline that systematically detects and corrects LLM-induced syntactic malformations. The sanitization layer addresses common failure modes such as: formatting artifacts, comment irregularities, literal format mismatches, and string boundary errors. Then a validation gate screens for residual anomalies like unmatched syntax elements, malformed numeric or string literal and catalogs violation patterns to inform iterative refinement. A critical recovery mechanism reconstructs essential structural declarations (class definitions, inheritance hierarchies) when the LLM omits them, preventing downstream assembly failures. The pipeline further performs corpus-level consistency audits, identifying structural violations across the entire variant population. By implementing domain-specific post-processing, the framework makes plausible code and into truly executable artifacts.

Built-in Recompiler: The framework integrates recompilation that transforms validated variants into executable artifacts, closing a gap in prior work that validates syntax but omits executable generation. By completing the source-to-executable pipeline, the framework enables empirical validation through dynamic analysis. Compilation failures feed back into the sanitization pipeline, creating iterative refinement where systematic errors are progressively eliminated.

4 Results and Discussion

We selected evaluation subjects comprising six Windows malware projects (C/C++), one Android source project (Java/Kotlin), and one Android APK binary to assess framework efficacy. Additionally, we retested RansomWar from the original foundational work to validate and calibrate against the authors' reported findings. The statistical details of the selected samples are describe in Table 1

To evaluate the variants generated by our framework we use the metric AV detection rate.

Anti-Virus (AV) Detection Rate: The AV detection rate measures the proportion of antivirus engines that flag a malware variant as malicious. Formally, for variant $\hat{M}_s$, the detection rate is $R_{\hat{M}_s} = \frac{|\hat{D}|}{|D|} \times 100\%$, where $\hat{D}$ is the set of detectors flagging the variant and D is the set of all available detectors. Multiple evaluation runs are performed and averaged to account for variability. **Lower detection rates indicate greater evasion success**.

Sample	Language	LOC	Files	Funcs	Type	Platform
He4Rootkit	C++	18,481	66	233	Rootkit	Win32
Internet-worm	C	197	1	2	Infector	Win32
SMMRootkit	C	19,246	11	111	Rootkit	Win32
RansomWar	C	1,126 6	14		Ransomware	Win32
GPUWinJelly	C	5,767	19	118	Ransomware	Win32
L3MONB0T	Smali	94,171	435	55		Android
FakeInst	Smali	86,284	74	-	SMSWare	Android (Binary)

Table 1: Brief description of malware projects selected for experiments.

Rates are measured using VirusTotal and Hybrid Analysis platforms that aggregate signature-based, heuristic, and ML-based detection approaches.

4.1 Results

Although we picked 6 windows malware projects written in C/C++, 1 android project and 2 android apk binaries for our experimental suit, some of the projects was completely non-functional after modification and some files could not be parsed by the pipeline. Given LLMs current limitations over large code bases it is somewhat expected. Finally, we present the results of 3 windows malware projects 1 android malware project and 1 decompiled apk project. Moreover due to time constraints we could not evaluate all the strategies for each of the malware. So, We conducted a systematic evaluation strategy to identify the most effective transformation approach for Windows malware.

Initial experiments on GPUWinJelly evaluated three candidate strategies: Strategy 1 (Code Optimization), Strategy 2 (Code Quality & Reliability), and Strategy 6 (Windows API-Specific Transformation). As shown in Figure 3, Strategy 6 demonstrated superior AV evasion performance, reducing AV detection from an initial baseline of 64% to 54% after eight modified functions, which is a 10% reduction that outperformed both Strategy 1 (which maintained 64-67% detection) and Strategy 2 (which remained relatively flat at 61-67%). Motivated by this we decided to focus exclusively on Strategy 6 for subsequent Windows malware evaluation. Consequently, Internet Worm and RansomWar variants were evaluated using only Strategy 6. Internet Worm (Figure 4) exhibited strong evasion characteristics, with AV detection declining from 55% at baseline to 46% at two modified functions—an 9% reduction—demonstrating that API transformation effectively evades detection in infector-class malware. RansomWar (Figure 5) achieved the most substantial improvements, reducing detection rates from 49% to 39% after only four modified functions, representing a 10% relative reduction in detection. This is also consistent with the results found for RansomWar by LLMalMorph, signaling our extended framework does not diminish the variant creation capability of the foundational baseline.

To prove our framework's (MPMALGEN) multi-platform malware variant generation capability, we tested our novel Android-specific transformation strategy (`strat_Android`) tailored to preserve Android component semantics while enabling evasion. It is the platform-level counterpart of Strategy 6 for windows malware. L3monB0T, an Android Remote Access Trojan implemented in Java/Kotlin, was evaluated using our custom Android strategy. As shown in Figure 6, the framework achieved modest but consistent evasion performance, maintaining detection rates between 35% and 45% across eight modified methods. The relatively flat detection rates (ranging from 43% baseline to 37% at maximum modification) reflect that the likely file level changes in such a large codebase is insufficient for evasion, since other files and API calls still can flag the project. It could be also due to the default security protocols in Android's permission-based detection model and the constraints imposed by preserving component lifecycle semantics.

To assess the framework's effectiveness on compiled Android artifacts, we evaluated FakeInst, an Android SMSware malware obtained as a decom-

piled APK binary. Our pipeline decompiled it into assembly level smali files with relevant resources and libraries. This was our second largest codebase tested with 86000+ lines of smali code. In this project we tested our multifile transformation feature which changes k number of functions in each of the smali files which contain $\geq k$ functions. Although such aggressive modification demonstrated huge decline in AV detection rate compared to the binary apk file binary representation, but it is likely due to the whole project unrecognizably changing and not being recompilable even. As shown in Figure 7, detection rates declined sharply from 51% at baseline to 1% after modifying 100 methods across all files and folders. Fixing and debugging is very difficult at such level for AI level patching, Though only 7 out of the 74 files have been flagged as having syntactical issues after the AI assisted patching. So, all in all, it demonstrates the possibility of being functional with additional human debugging effort.

Limitations: Our experiments were constrained to ten malware samples across three categories (six Windows, one Android source, one APK binary) and results are limited to only five malware projects, limiting statistical generalization to broader malware populations. Android semantic preservation remains challenging due to strict component lifecycle requirements and permission model constraints, resulting in almost no evasion gains compared to Windows variants. Moreover, our framework requires manual debugging intervention for complex multi-file transformations, particularly for platform-specific API substitutions, preventing fully automated variant generation at scale. Results are only evaluated with one metric which is mostly static analysis dependent. Functionality preservation and hybrid analysis results have not been explored due to time constraints. Moreover error free recompilation to binary for APK files does not work as of now and requires manual work, even if feasible. So, static analysis comparison between APK and uncompilable file is somewhat misleading. Also test results with only two models does not show the code comprehesnion and alteration capability of huge LLM landscape. Task specific fine-tuning is also unexplored and have not been investigated by any prior works as well.

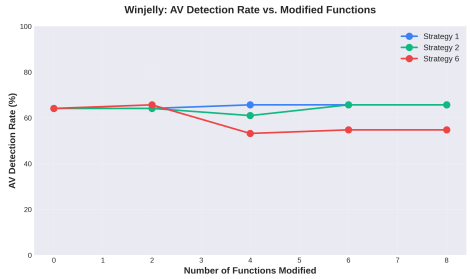

Figure 3: GPUWinJelly: AV Detection Rate vs. Modified Functions across Strategies 1, 2, and 6.

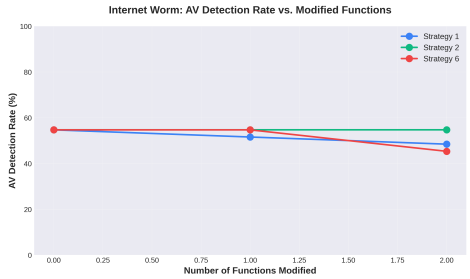

Figure 4: Internet Worm: AV Detection Rate vs. Modified Functions using Strategy 6.

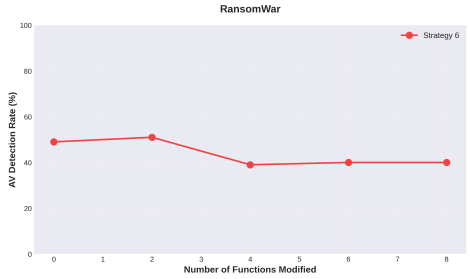

Figure 5: RansomWar: AV Detection Rate vs. Modified Functions using Strategy 6.

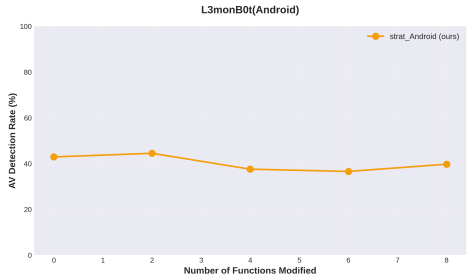

Figure 6: L3monB0T: AV Detection Rate vs. Modified Methods using Android Strategy.

References

Md Ajwad Akil, Adrian Shuai Li, Imtiaz Karim, Arun Iyengar, Ashish Kundu, Vinny Parla, and Elisa Bertino. 2025. Llmalmorph: On the feasibility of generating variant malware using large-language-models. *arXiv preprint arXiv:2507.09411*.

M. Botacin. 2023. Gpthreats-3: Is automatic malware

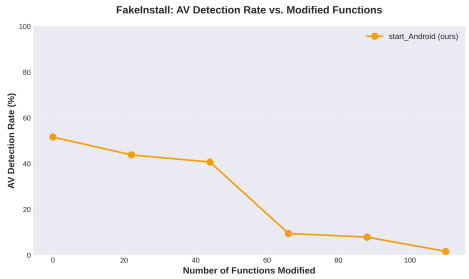

Figure 7: FakeInst: AV Detection Rate vs. Modified Methods using Android Strategy.

generation a threat? In *Proceedings of the IEEE Security and Privacy Workshops (SPW)*, pages 238–254.

J. Choi, D. Shin, H. Kim, J. Seotis, and J. B. Hong. 2019. Amvg: Adaptive malware variant generation framework using machine learning. In *Proceedings of the 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 246–262.

X. Ling, Z. Wu, B. Wang, W. Deng, J. Wu, S. Ji, T. Luo, and Y. Wu. 2024. A wolf in sheep's clothing: Practical black-box adversarial attacks for evading learning-based windows malware detection in the wild. In *Proceedings of the 33rd USENIX Security Symposium*, pages 7393–7410.

K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre. 2021. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 744–758.

J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao. 2017. Impeding behavior-based malware analysis via replacement attacks to malware specifications. *Journal of Computer Virology and Hacking Techniques*, 13:193–207.

Y. Qiao, W. Zhang, Z. Tian, L. T. Yang, Y. Liu, and M. Alazab. 2022. Adversarial malware sample generation method based on the prototype of deep learning detector. *Computers & Security*, 119:102762.