



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



Vilnius
University

Gabriel Pereira

Code Obfuscation and Deobfuscation Techniques using LLMs

Master Dissertation

Master's on International Cybersecurity and Cyberintelligence

Work developed under the guidance of:

Gabriel Maciá

June 2026

Resumo

Este trabalho investiga o potencial e as limitações da aplicação de modelos de aprendizagem automática em processos de ofuscação e deofuscação de código. Embora os Modelos de Linguagem de Larga Escala representem o foco central das tendências tecnológicas atuais, a análise do estado da arte e consequentes testes demonstra que a sua utilização prática no desenvolvimento de funções maliciosas específicas — tais como algoritmos de geração de domínios (DGA) e técnicas de packing — enfrenta restrições severas. Estas limitações decorrem da elevada exigência computacional e da complexidade inerente a estas arquiteturas de grande porte.

Como alternativa, esta dissertação propõe o recurso a estruturas neuronais mais compactas e modulares como camadas de embedding, Transformers simplificados ou Redes Neuronais mais básicas. Por apresentarem uma pegada computacional reduzida, estas arquiteturas alinham-se com o paradigma de integração de Machine Learning como componente de software nativo, introduzindo dinâmicas capazes de contornar os métodos tradicionais de análise e *reverse engineering*. Embora os testes experimentais realizados sejam preliminares e os resultados se apresentem ainda incompletos, a evidência empírica recolhida sugere a viabilidade destas abordagens, abrindo caminho para futuras investigações no desenvolvimento de mecanismos alternativos de proteção e obfuscação.

Abstract

This work investigates the potential and limitations of applying machine learning models to code obfuscation and deobfuscation processes. Although Large Language Models (LLMs) represent the central focus of current technological trends, state-of-the-art analysis and subsequent empirical evaluation demonstrate that their practical deployment for specific malicious functions—such as Domain Generation Algorithms (DGAs) and packing techniques—faces severe constraints. These limitations stem from the high computational demands and resource overhead inherent to these large-scale architectures. As a lightweight alternative, this dissertation proposes the utilisation of more compact and modular neural structures, such as embedding layers, simplified Transformers, or simple multi layer perceptrons. By presenting a significantly reduced computational footprint, these architectures align with the paradigm of integrating machine learning as a native software component, introducing structural dynamics capable of circumventing traditional software analysis and *reverse engineering* methods. Although the experimental tests conducted are preliminary and the results remain incomplete, the gathered empirical evidence suggests the viability of these alternative approaches, paving the way for future research into novel mechanisms for code protection and obfuscation.

Index

1	Introduction	1
1.1	Introduction	1
1.2	Formalization	1
1.3	Current State-of-the-Art	2
1.3.1	Obfuscation	2
1.3.2	Deobfuscation techniques	5
1.4	Contextualisation	6
2	Literature Review	9
3	Methodological Approach	11
3.1	What defines a good obfuscation	11
3.2	Obfuscation with LLMs	13
4	Work Done	15
4.1	Architecture and Design	15
4.2	LLM as a Calculator	17
4.3	MLP as a Calculator	21
4.4	Domain Generation Algorithm	22
4.5	Neural Hash Map	26
4.6	MLP as Packer	26
4.7	Transformer model as Packer	33
4.8	Custom Remote Access Trojan	34
5	Conclusion	39

Figures Index

Workflow of LLM as Code Architecture	15
File Composition of model.gguf	17
MLP as Sum architecture	21
MLP as DGA architecture	23
MLP DGA Selection Process	24
MLP as DGA architecture	27
MLP Packer Performance	28
<i>File 1</i> VT score	30
<i>File 2</i> VT score	30
<i>File 2</i> AV Detections	30
NPacked Conficker Byte level Entropy	31
NPacked Cacheset64Wrapper Byte level Entropy	32
NPacked file Byte level Entropy an expanded Network	33
MLP vs NNCP Extracting Performance	34
simple RAT architecture	35
simple RAT Entropy per Section	36
Connection to C2 by custom RAT	37
Control (top) vs Custom (bottom) Sample analyzed by VT 3.0	37

Tables Index

Classification of Obfuscation Techniques	7
Classification of Deobfuscation Techniques and Tools	8
Pass rate and Execution time for a Qwen2ForCausalLM architecture model	19
Pass rate and Execution time for a Qwen2ForCausalLM expanded architecture model	19
Pass rate and Execution time for Qwen2.5-3B-Instruct using Q4_K_M quantization	20
Pass rate and Execution time for sum calculator MLP	22
Pass rate and Execution time for DGA MLP	24
Hostname Validity for Out-Of-Bounds Seed Spaces	25
Comparison: Traditional Data Structures vs. Implicit Neural Data Representations	26
Computational Runtime Scaling for Packing and Extraction Operations - NPacker	28
Structural and Complexity Comparison: Baseline UPX vs. Neurally Packed Payload	29
UPX vs NPacker Entropy comparison	31
Computational Runtime Scaling for Packing and Extraction Operations - NNCP	33

1 Introduction

1.1 Introduction

While the established state-of-the-art obfuscation frameworks and analytical tools form a robust foundation for malware analysis, they reflect a historical arms race dynamic that has reached a temporary equilibrium. Modern analysis paradigms are heavily optimized to recognize and unpack traditional signatures and conventional obfuscation patterns. Consequently, relying entirely on known baselines creates a critical blind spot against next-generation evasion strategies. To proactively evaluate the limitations of contemporary defensive detection architectures, it is vital to venture past traditional packing mechanics. This work addresses this exact boundary by investigating novel obfuscation paradigms and stealth techniques, mapping out how LLM based methods or better and simpler ML model alternatives can bypass state-of-the-art analytical frameworks.

1.2 Formalization

Code obfuscation and deobfuscation are opposing processes related to software engineering and cybersecurity, they can be seen as different sides of the same coin. Code obfuscation refers to the transformation of a program into a version that preserves its functionality while making its logic, structure, or data representations difficult to understand, analyze, or reverse engineer. Formally, an obfuscation transformation can be seen as a semantics-preserving mapping $f(P)$:

$$\begin{aligned} f(P) &= P' \\ \forall x \in \mathcal{X}, \quad P(x) &= P'(x) \end{aligned} \tag{1.1}$$

where P is the original program and P' is the obfuscated one exhibiting reduced readability and analyzability. In contrast, deobfuscation aims to recover, approximate, or reconstruct the original program logic from an obfuscated representation. Deobfuscation techniques attempt to undo structural transformations, simplify complex expressions, and reveal hidden control or data flows, often without access to the original source code.

Code obfuscation can be deployed for various objectives. Firstly, within a benign paradigm, it represents a cost-effective and highly efficient methodology for safeguarding intellectual property (IP) [65]. For commercial enterprises whose core valuation resides in proprietary software architectures—such as large language models (LLMs) or software-as-a-service (SaaS) platforms, the implementation of robust protective measures is imperative [46, 73]. A prominent case study of IP preservation utilizing commercial-grade digital rights management (DRM) utilities is observed within the interactive entertainment industry. Denuvo Software Solutions GmbH has provisioned specialized binary obfuscation services since the commercial release of FIFA 15. [30, 12].

In reality, these tools are of dual use. The same frameworks engineered to safeguard proprietary software are frequently abused to provide stealth and detection evasion for malicious software families. For example, while software **packers** were historically deployed by legitimate distributors to minimize storage usage and optimize distribution efficiency, contemporary usage has shifted. In the modern threat landscape, these mechanisms are systematically leveraged by malware developers as an obfuscation layer to mutate executable signatures and encapsulate malicious payloads, thereby obstructing static analysis and signature-based detection [56].

It is critical to delineate the opposing objectives governing these two software protection paradigms. Within benign commercial deployment, software vendors utilize obfuscating transformations to defend against man-at-the-end (MATE) attack vectors - Principally reverse engineering and tampering, with the explicit goal of preserving proprietary assets. Conversely, malware authors execute functionally identical transformations with a fundamentally distinct objective: they induce a blind spot in automated security controls, specifically targeting evasion [14].

1.3 Current State-of-the-Art

Programming languages will be classified according to [15] as:

- *Native languages* - They are compiled to and distributed as native code binaries, for example C/C++.
- *Managed languages* - These are typically compiled and distributed as bytecode to be executed in a managed environment, for which enough symbolic information needs to be included. Java or C# are good examples.
- *Scripting languages* - e.g. JavaScript or PHP and are distributed as source code mostly for web applications.

Languages that differ from these previous definitions will be referred to as *No domain* languages. Moreover, there are three different categories for code considered. First, *Source code*, which can be rewritten, for example, with the C-to-C rewriter Tigress [25]. Secondly, *Native code* is in the form of assembler or binary object code and can be transformed, for example, with link-time binary rewriters such as Diablo [1]. Finally, protections can be deployed on *Intermediate code* formats. These formats can be either compiler intermediate representations such as LLVM's bitcode, or the bytecode distribution formats of managed language, such as Java bytecode or the C Intermediate Language (CIL).

1.3.1 Obfuscation

There are multiple obfuscation techniques, each one of them encompassing different features and being applied to specific scenarios.

Data-oriented transformations attempt to hide semantically meaningful information embedded in the program. A common example is **string encoding**, where literal strings are stored in encoded form (e.g. Base64, hexadecimal, ROT13, or XOR with a key) and decoded only at runtime. This prevents static analysis tools from easily extracting

sensitive information such as API names, command strings, configuration parameters, paths or URLs [22]. Similarly, instead of obfuscating strings, **mixed boolean-arithmetic (MBA)** obfuscation replaces simple arithmetic or logical expressions with more complex, yet mathematically equivalent, combinations of Boolean and arithmetic operations [55]. For example,

$$a + b = (a \oplus b) + 2 \times (a \& b). \quad (1.2)$$

Such transformations significantly increase symbolic complexity and hinders simplification by automated reasoning engines and SMT solvers [36]. Destinations of jump instructions can be also obfuscated encoding their target address.

Another widely used technique is the insertion of **opaque predicates**. These are conditional expressions that always evaluate to a constant boolean value at runtime, but are constructed in a way that makes this property difficult to prove statically. By introducing opaque predicates, the obfuscator creates misleading branches and complicates control-flow analysis [13, 74]. This can be imaged by the following snippet:

Listing 1.1: Opaque Predicate technique in C

```
1 int main() {
2     int x = 3;
3     int y = 5;
4     int z = x * y;
5
6     // Opaque predicate invariant: z == 15,
7     // therefore this if statement will always be true
8     if (z == 15) { ...
```

At the structural level, **dead code insertion** consists of adding instructions or entire code blocks that do not affect the semantics of the program. Unlike simple NOP instructions, dead code may involve complex computations or artificial branches, thereby increasing the apparent complexity of the control-flow graph and defeating signature-based detection mechanisms [47].

For control-flow transformations, the most well-known is **code flattening**. In this approach, the structured control flow of a function is replaced by a centralized dispatcher loop governed by a state variable. All basic blocks are reorganized under this dispatcher, typically implemented as a `switch` statement or jump table. Instead of nested constructs, execution proceeds through state transitions, making control-flow recovery significantly more difficult [67, 39].

Some techniques rely on runtime mechanisms to further obscure program behavior. **Dynamic code generation** constructs executable code at runtime (e.g. through `exec` or `eval` statements), preventing static

inspection of the final instruction stream. Similarly, **reflection** and dynamic API resolution outsource control-flow decisions to runtime environments, thereby obscuring call graphs and dependencies. There is also the possibility of replacing conditionals with **exception programming** where exception handlers are leveraged [69].

An even more advanced technique is **virtualization-based obfuscation**. In this approach, selected portions of the program are translated into a custom bytecode executed by an embedded virtual machine (VM). The VM defines a proprietary instruction set with undocumented semantics, which means that the protected logic is interpreted rather than executed natively. Reverse engineering therefore requires understanding both the custom instruction set and the interpreter implementation, substantially increasing analysis complexity.

Packing is also a common integrity option as it transforms a program into a compressed or encrypted representation and embeds it within a small loader stub. When executed, the stub restores the original code directly in memory and transfers control to it. This prevents static analysis tools from accessing the actual payload without emulating or executing the unpacking routine. Modern packers often combine compression, encryption, and anti-analysis techniques [3, 17].

Metamorphism has been observed throughout the years as another great way of avoiding signature detection and delay the process of obfuscation. It consists of the code rewriting itself, commonly through a mutation engine inside the program.

Whenever someone is reverse engineering or analyzing a program, most probably controlled environments such as virtual machines and emulators will be used to achieve flexibility and replication. Some malware families [20] detect if they are inside a sandboxed environment, and if confirmed, they hide their malicious intent and appear as goodware. It is also a common practice for malware programs to start executing their payload not at start of execution but rather after some time [33].

In practice, the desired effectiveness of obfuscation relies on layering multiple techniques across abstraction levels. Lightweight data transformations hinder automated scanning, structural modifications complicate static control-flow reconstruction, and binary-level protections defend against deeper reverse engineering. The strength of modern obfuscation thus lies not in any single technique, but in their strategic and combined application.

The techniques mentioned above can be classified as follows in table 1. The criteria for selecting the *type* column is the practical effect the transformation has on the program. For example, encoding the target of a jump instruction is not considered a data obfuscation, but rather a control flow obfuscation, since it tries to hide the program's flow.

Both open-source and commercial implementations provide diverse capabilities.

To obscure software profiles from external review, tools range from structural source modifiers like **Tigress** and compiler-integrated engines like **OLLVM**, to commercial binary virtualization frameworks like **VMProtect**, **Themida**, and **Code Virtualizer**, complemented by targeted language protectors including **ProGuard**, **Allatori**, **Dash0**, **Zelix** **Klassmaster** [15]. **UPX** (The Ultimate Packer for eXecutables) remains as one of

the most mentioned packers [53, 48] and this tool is leveraged by a large quantity of malware executables.

1.3.2 Deobfuscation techniques

Mapping the specific vectors that target obfuscation mechanisms is critical, providing the foundational baseline required for rigorous obfuscation quality and resilience testing.

- **Static Disassembly and Control-Flow Reconstruction** - Translates raw binary machine code into readable assembly language without executing the payload. This baseline process maps out the structural skeleton of the binary, detailing functions, basic blocks, and jump targets [49].
- **Taint Analysis** - Taint analysis tracks how data (like user input or sensitive values) flows through a program. Taint analysis helps deobfuscating by focusing only on data that truly influences program behavior, ignoring irrelevant instructions [24].
- **Symbolic Execution** - Symbolic execution treats program inputs as symbolic variables instead of specific values. Rather than running the program step by step with real input, it explores many possible execution paths. If the program is big, this method will not be effective due to program explosion [5].
- **Program Synthesis-based Deobfuscation** - Rather than directly analyzing the obfuscated code, a synthesis engine generates a simpler program that behaves the same on all inputs. This produces a clean pseudo-version of the intended logic [4].
- **Dynamic Execution and Behavioral Analysis** - Runs the binary inside an isolated environment to observe and record its actual runtime activity rather than relying on static code patterns. By logging system changes (process creation, registry modifications, network traffic) this methodology builds an interactive profile of the binary's intent. This form of dynamic analysis is standardly automated using specialized live sandboxing platforms, including interactive cloud tools like ANY.RUN and automated orchestration engines like CAPE Sandbox (or its open-source predecessor Cuckoo Sandbox). Other simpler tools include Wireshark and RegShot [16].

Numerous deobfuscation tools and binary analysis frameworks exist to counter protection and obfuscation. Binary parsing assets like Jadx and Dex2Jar revert compiled execution structures back into human-readable source. Structural analysis platforms include IDA Pro and objdump, comparative tools like BinDiff, dynamic multi-layer engines like OllyDbg, and specialized mobile framework parsers such as APKTool and AndroGuard. Advanced algorithmic verification utilizes symbolic execution and logic solvers, which pair automated flaw discovery engines like Angr, Triton, and KLEE with mathematical constraint solvers like Z3, mathematical deobfuscators like Arybo, and structural pattern recovery suites like Syntia. Table 2 provides a summary of this information and other tools used with each end in mind.

1.4 Contextualisation

All these obfuscation and deobfuscation methodologies, specifically the latter, will be systematically considered throughout this paper. They will be actively applied not only to facilitate the measurement of core metrics to be defined further ahead but also to extract valuable security intelligence and actionable insights regarding the underlying behavior, intent, and structural properties of the target software. By leveraging these techniques simultaneously, this work bridges the gap between theoretical evasion capabilities and empirical analysis, using defensive deobfuscation frameworks to validate the real-world resilience and stealth metrics of the proposed implementations.

Classification of Obfuscation Techniques

Obfuscation Technique	Description	Type
String Encoding	Encode strings so they are stored in encoded form and decoded only at runtime. This includes data such as API names, URLs, or commands.	Data
Literal / Constant Encoding	Replace literal constants with encoded or computed values that are reconstructed during execution.	Data
Arithmetic Encoding (MBA)	Replace simple arithmetic operations with complex combinations of Boolean and arithmetic expressions that are mathematically equivalent.	Data
Opaque Predicates	Insert conditional branches that always evaluate to a constant value at runtime but are difficult to prove statically.	Control Flow
Control Flow Flattening	Replace structured control flow with a dispatcher loop controlled by a state variable that determines which block executes next.	Control Flow
Jump / Branch Encoding	Obfuscate the destination of jumps by computing branch targets indirectly or through encoded expressions.	Control Flow
Dead Code Insertion	Add instructions or blocks that are never executed but increase the complexity of the control-flow graph.	Control Flow
Virtualization-based Obfuscation	Translate parts of the program into a custom bytecode executed by a proprietary virtual machine embedded in the program.	Control Flow
Dynamic Code Generation	Generate executable code at runtime (e.g., through JIT compilation or runtime decoding).	Control Flow
Reflection	Resolve function calls dynamically at runtime to hide the real dependencies and call graph.	Control Flow
Environment Checking	Analyze the environment and modify behavior based on results	Integrity
Packing	Compress or encrypt the program and unpack it at runtime using a loader stub.	Integrity
Metamorphism	Continuously rewrite or mutate the code so that each instance of the program looks different while preserving behavior.	Integrity

Classification of Deobfuscation Techniques and Tools

Deobfuscation Technique	Description	Associated Tools
Static Disassembly and Control-Flow Reconstruction	Translates machine code to assembly without execution to map functions, basic blocks, and jump targets.	IDA Pro, Ghidra, Binary Ninja, Radare2, Rizin, objdump, BinDiff, APKTool, Cutter
Taint Analysis	Tracks how data flows through a program to isolate instructions that truly influence behavior, filtering out dead code noise.	Triton, PANDA, Intel PIN, Taintgrind
Symbolic Execution	Treats inputs as symbolic variables to mathematically evaluate multiple execution paths; vulnerable to path explosion.	angr, Manticore, Triton, KLEE
Program Synthesis-based Deobfuscation	Treats code blocks as black boxes and automatically generates a simplified, clean program that behaves identically on all inputs.	Syntia, QSynthesis
Dynamic Execution and Behavioral Analysis	Executes code in isolated environments to log live system modifications, process creation, registry tweaks, and network traffic.	ANY.RUN, CAPE Sandbox, Cuckoo Sandbox, Frida, x64dbg, Wireshark, RegShot, Procmon, VMRay, VirusTotal (AV)

2 Literature Review

Code obfuscation is not a novel concept; historically, it has been widely employed by legitimate developers to protect intellectual property from reverse engineering [46, 65, 73, 30, 12]. Concurrently, threat actors have heavily leveraged these same techniques to mask malicious payloads and evade standard security detection [18, 21]. To navigate this landscape, foundational studies such as the one by De Sutter et al. [15] provide comprehensive evaluation methodologies, cataloguing the most prevalent obfuscation and deobfuscation paradigms currently utilized by researchers and practitioners.

Within the realm of traditional code protection, source-to-source obfuscators such as Tigress [25] remain foundational, offering a comprehensive suite of transformation techniques. These tools systematically alter the abstract syntax tree (AST) and control flow of source code through complex methodologies—such as control-flow flattening, the injection of non-trivial opaque predicates, and mixed boolean-arithmetic (MBA) formulations [20, 69, 67, 36, 9, 13, 74, 22, 39, 55]. Parallel to these source-level modifications, executable packers continue to serve as a highly effective, binary-level resource. By compressing, encrypting, and wrapping compiled binaries with a custom decompression stub, packers effectively mask the executable’s true semantic intent, thereby hindering static analysis tools and raising the barrier for structural detection engines [53, 48, 3, 17].

The efficacy of these obfuscation methodologies is typically evaluated through a combination of structural metrics and practical resilience testing. Quantitative metrics—ranging from basic entropy analysis to advanced semantic elasticity and BERT-based similarity detection—provide a baseline for measuring structural divergence [38, 2, 29, 59, 62]. Furthermore, to be deemed robust, an obfuscator must successfully hinder standard reverse-engineering toolchains [23, 40, 49, 31, 8] and withstand rigorous, automated deobfuscation techniques, such as symbolic execution, taint analysis, and program synthesis [5, 57, 4, 49, 24, 27].

As the research focus shifts towards machine learning interventions, evaluating standard Large Language Model (LLM) architectures and their respective deployment ecosystems (e.g., GGUF, ONNX, and Safetensors) reveals significant practical limitations for stealthy, on-device obfuscation [52, 11, 37, 43, 10, 70]. Driven by the severe computational overhead of these large frameworks, a logical pivot is made towards more efficient, algorithmic alternatives. Architectures such as LSTMs and neural-based data compression networks demonstrate superior viability for deterministic logic tasks compared to standard LLMs [6, 7, 68, 72, 26]. To validate this lightweight neural paradigm, subsequent empirical evaluations incorporate varied threat vectors—including Remote Access Trojans (RATs) and NT-level API hooking techniques—tested dynamically against established analysis pipelines [40, 19, 54, 63]. Furthermore, established real-world applications of Implicit Neural Representations (INRs) provided a foundational architectural framework, laying down the structural paradigms required to implement the proposed neural obfuscation mechanisms [32, 71].

To conduct robust empirical evaluations and ensure some level of real-world applicability of the proposed neural architectures, samples were systematically curated from established malware repositories and open-source platforms. Notably, the VX Underground database [66] served as a primary resource, providing a comprehensive

collection of contemporary threat source codes and compiled binaries. This dataset was further supplemented by theZoo [61], a well-documented repository of live malware designed explicitly for academic research and defensive analysis.

3 Methodological Approach

3.1 What defines a good obfuscation

In order to define a good obfuscation, its objective and metrics that evaluate the code need to be established, as well as what the latter imply. At the beginning of this document the process of obfuscation was defined as a function that transforms a program P into another program P' with the same behaviour. Therefore the *Pass Rate* is and will be the first metric to look at after any obfuscation process. Following that are two baseline metrics for software performance, the *Execution Time Difference* and *Code Expansion Ratio*. The three metrics are defined below.

a) Pass Rate

The *Pass Rate* measures the degree to which the obfuscated code preserves the functional behavior of the original implementation. It is defined as the fraction of test cases for which the outputs (or desired side effects) of the original and obfuscated functions are identical. This metric represents **the most critical requirement for successful obfuscation**, namely semantic equivalence, and it is used to assess the correctness of program transformations.

$$P = \frac{1}{n} \sum_{i=1}^n I(f_{\text{orig}}(t_i) = f_{\text{obfs}}(t_i)) \quad (3.3)$$

Higher values of P indicate better preservation of functionality.

b) Execution Time Difference

The *Execution Time Difference* evaluates the runtime overhead introduced by obfuscation. It is calculated as the average difference in execution time between obfuscated and original code across all test cases.

$$\Delta T = \frac{1}{n} \sum_{i=1}^n (T_{\text{obfs},i} - T_{\text{orig},i}) \quad (3.4)$$

Values close to zero indicate minimal performance impact, while positive values reflect execution slowdowns.

c) Code Expansion Ratio

The *Code Expansion Ratio* quantifies the relative increase in code size caused by obfuscation. It is computed as the ratio between the number of lines in the obfuscated code and the number of lines in the original code.

$$E = \frac{\text{Lines}_{\text{obfs}}}{\text{Lines}_{\text{orig}}} \quad (3.5)$$

While an increased expansion ratio may contribute to resistance against reverse engineering, it also reflects reduced efficiency and maintainability. Therefore, lower values of E are preferred when obfuscation strength is preserved.

As for the objective, the operational objective for these obfuscation techniques will be malware-oriented. Consequently, any reference to code implies the analysis of disassembled native code rather than source code representations.

Based on a review of literature [47, 2, 59, 42, 23], a definitive suite of metrics was selected to quantify the efficiency and structural overhead of the applied obfuscation techniques.

d) Cyclomatic Complexity Change

The *Cyclomatic Complexity Change* measures the structural impact of obfuscation on control flow. It is defined as the difference between the McCabe cyclomatic complexity of the obfuscated and original code.

$$\Delta CC = CC_{\text{obfs}} - CC_{\text{orig}} \quad (3.6)$$

Cyclomatic complexity is computed by counting decision points in the abstract syntax tree, including constructs such as `if`, `for`, `while`, and `lambda`. For the assembly language, later in this paper, variants of `jmp` and `loop` will be used in this calculation. Larger absolute values of ΔCC indicate more substantial control-flow transformations, translating into a slower execution time.

e) Cosine Similarity

Cosine Similarity measures the lexical similarity between the original and obfuscated code by computing the cosine of the angle between their vector representations, such as TF-IDF (Term Frequency-Inverse Document Frequency) or token-frequency vectors. This metric is insensitive to absolute code length and is effective at identifying structural resemblance despite superficial transformations.

$$CS(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} \quad (3.7)$$

Here, A and B denote the vector representations of the original and obfuscated code, respectively. Higher cosine similarity values indicate stronger lexical similarity.

f) Character-wise Entropy Change

Character-wise entropy measures the change in Shannon entropy of character distributions between the original and obfuscated code. Increased entropy reflects higher randomness and reduced predictability, which correlates

with decreased readability. Furthermore, this metric serves as a fundamental feature within automated malware detection pipelines, where elevated statistical thresholds systematically trigger heuristic flags.

For a code fragment X , Shannon entropy is defined as:

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \log_2 p(x) \quad (3.8)$$

where $p(x)$ denotes the empirical probability of character x in the code.

The character-wise entropy change is computed as:

$$\Delta H = H_{\text{obfs}} - H_{\text{orig}} \quad (3.9)$$

A positive value of ΔH indicates increased randomness in the obfuscated code.

g) Evasion Ratio

Finally, a metric for quantifying malware detection evasion must be formulated to empirically evaluate the defensive circumvention efficiency of the applied obfuscation techniques. For each test T and malware sample m , the *Evasion Ratio* S will be accessed by the formula:

$$S = \frac{\sum_{T \in \mathcal{T}} \sum_{m \in \mathcal{M}} (1 - T(m))}{|\mathcal{T}| \times |\mathcal{M}|} \quad (3.10)$$

where the detection outcome for an individual test-sample pair is given by:

$$T(m) = \begin{cases} 1, & \text{if } T \text{ detects } m, \\ 0, & \text{if } m \text{ evades } T. \end{cases} \quad (3.11)$$

3.2 Obfuscation with LLMs

As the title would suggest, the intuitive initial approach to LLM-driven obfuscation involves directly feeding raw source code or compiled executables into a large language model with instructions to apply standard obfuscation techniques. Existing frameworks can already exploit generative pipelines, either by subverting an LLM to inject malicious payloads into user-requested scripts [60]. However, this end-to-end framework introduces significant operational limitations. State-of-the-art obfuscation tools are fundamentally algorithmic in nature, relying on deterministic compiler passes and formal mathematical abstractions to guarantee functional preservation. Consequently, large language models are inherently not suited for these highly complex, low-level execution tasks,

as their underlying probabilistic architecture is incapable of enforcing rigid semantic constraints required for dependable binary transformations [72, 26].

The idea of the novel technique to be introduced with this paper is to leverage the black-box nature of LLMs and replace certain software artifacts with such models. For example, in case of arithmetic expressions, instead of using default language-specific operators, certain calculations would be performed by an LLM backend.

Listing 3.2: Example of LLM as software

```
1 int main() {  
2     // Normal execution  
3     int x = 10 / 2 + 5;  
4  
5     // LLM backend  
6     int y = llm_arithmetic("10 / 2 + 5");  
7     ...  
}
```

Under the lens of automatic static analysis, the analyzer would have to trace-back the function definition, and at the end of the line there is a file containing only a collection of float point values - an LLM. This technique could fall under the dynamic code generation category as explained in the *Introduction* chapter, but differs in the way that the core logic is stored under the form of neuronal weights, bias and a mandatory architecture for the program to know how to run such data. Using this method, simple functional behaviours could be replicated **without writing a single line of code**, which in itself is a powerful abstraction capable of defeating the standard deobfuscation paradigms.

A conceptual analogy is that of a CPU systematically executing its primary operational loop while periodically offloading arithmetic problems to the ALU (Arithmetic Logic Unit) for processing.

The next section describes the methods used to implement such strategies and test cases.

4 Work Done

Description of the work done. The structure of this chapter is very case-dependent. However, there should frequently be sections on design, implementation, and testing/verification.

4.1 Architecture and Design

In order to use the Large Language Models as executable code proxies, the system architecture consists of a minimalistic two-part implementation framework consisting of an execution loader and the underlying parametric data model as presented in Figure 1.

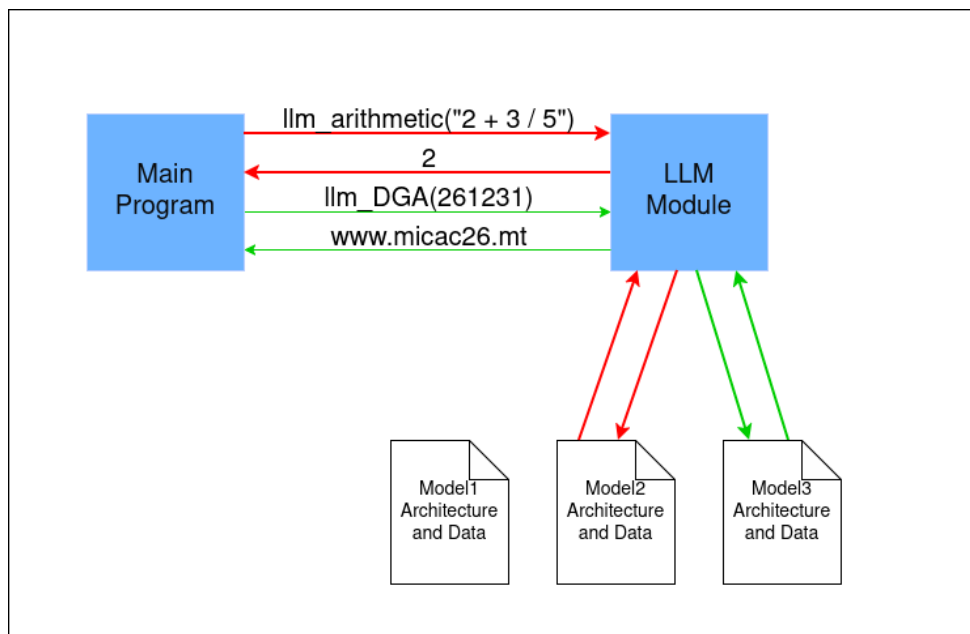


Figure 1: Workflow of LLM as Code Architecture

The execution loader must dynamically switch between different models at runtime in order to fulfill the function required. These model components are partitioned into two distinct segments: an explicit metadata schema that dictates parsing parameters for the host engine, and a serialized weight tensor array containing the underlying network parameters.

To store both the structural metadata schema and the underlying network weights, four model serialization formats were evaluated: PyTorch Checkpoints (.pt), Safetensors (.safetensors), Open Neural Network Exchange (.onnx), and GGML Unified Format (.gguf). Each format presents distinct architectural trade-offs regarding security, metadata encapsulation, and execution latency.

a) PyTorch Checkpoints (.pt)

The ubiquity of the PyTorch library makes it the default environment for training Deep Neural Networks (DNNs) and Large Language Models. A standard deployment pipeline exports the optimized model using the TorchScript compiler `torch.jit.script(model).save()`, generating a .pt container designed for serialization. However, from a deployment perspective, this format introduces big architectural and operational challenges. A TorchScript file is an un-optimized ZIP archive that aggregates numerical tensor blocks, configuration metadata, and operational code serialized via Python’s pickle framework alongside torch specific artifacts.

The primary limitation of this encapsulation scheme is its heavy execution dependency chain. Deserializing and executing a .pt or TorchScript binary requires a complete, resource-heavy installation of the PyTorch framework, necessitating a Python runtime environment to host the execution loop, therefore compromising cross-platform portability.

b) Safetensors (.safetensors)

Although the safetensors format is widely used for its support of deterministic zero-copy memory mapping and robust mitigation against arbitrary code execution vulnerabilities [10], its adoption introduces a different problem. Because the design criteria of safetensors focus exclusively on secure tensor serialization rather than execution-graph encapsulation, it leaves out vital structural metadata such as built tokenization rules, platform-specific quantization tables, and general network hyperparameters. Consequently, utilizing safetensors would require the creation of a packaging architecture from scratch.

c) The ONNX (.onnx) and GGUF (.gguf) formats

The architectural evaluation of these two model serialization formats reveals that both ONNX and GGUF offer the critical advantage of decoupling machine learning inference from heavy Python environments by utilizing native C++ runtimes, enabling faster, platform-agnostic execution on target host hardware. GGUF shines as an all-in-one container engineered specifically for Large Language Models, storing the model weights directly alongside vital execution metadata like the tokenizer, vocabularies and context hyperparameters. As for ONNX, enabling it for cross-platforming results in an heavy framework, and it also needs to be setup with Python. [37, 43, 52, 11].

Moreover, the open-source availability of the llama.cpp repository (<https://github.com/ggml-org/llama.cpp>) fundamentally bridges the gap between high-level model training and low-level system deployment [41]. By providing native, python-based conversion scripts, the ecosystem enables developers to ingest complex transformer model weights directly from PyTorch training architectures, compress them via advanced sub-byte quantization algorithms, and serialize them cleanly into the GGUF specification [34]. This transition allows the execution loop to run entirely inside a pure C/C++ inference runtime, bypassing the overhead of interpreter layers

and significantly accelerating raw memory-bound tensor math. Beyond simplifying the engineering lifecycle, this aggressive quantization and dependency-free compilation strategy allows large language model architectures to be executed smoothly on standard, unaccelerated consumer hardware. From a security perspective this low-overhead inference satisfies the design parameters required by malware, as it allows advanced, quantized neural processing architectures to run into standard personal hardware environments. Picture 2 depicts the underlying structure of such format.

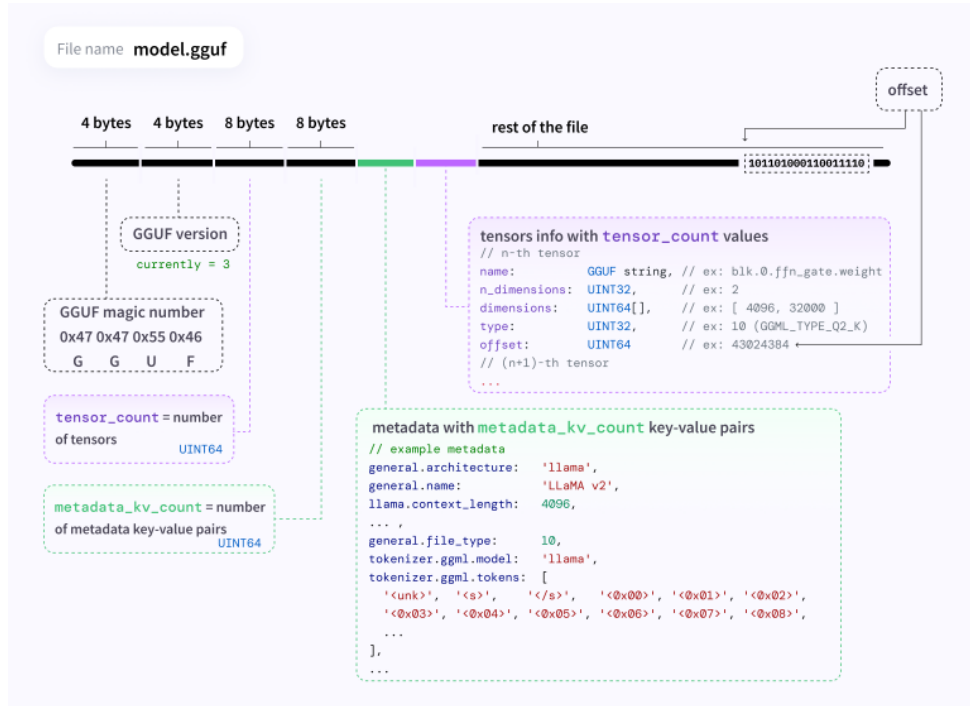


Figure 2: File Composition of model.gguf

4.2 LLM as a Calculator

To validate the architectural framework, the initial phase of empirical testing focused on training a Large Language Model to operate as a deterministic arithmetic solver, as schematically illustrated in Figure 1. In accordance with the engineering design finalized in the preceding section, the unified pipeline splits training from deployment. The initial training phase leverages the high-level abstraction layers of the PyTorch framework implemented within a native, cloud-hosted Python environment utilizing a Google Colab T4 graphics processing unit (GPU). Upon structural convergence, the finalized neural graph and network parameters are serialized and exported as a singular GGUF binary asset optimized for low-latency endpoint execution. Downstream evaluation and inference testing were conducted locally on a standard consumer-grade laptop equipped with an Intel Core i7-9750H CPU (operating at 2.60GHz) and an NVIDIA GeForce GTX 1660 Ti mobile GPU (6 GB GDDR6 VRAM) integrated with

CUDA compilation capabilities (Version 13.2).

a) Building from Scratch

The process began by selecting a model architecture inherently compatible with GGUF conversion, which led to the deployment of an untrained, custom-configured Qwen2ForCausalLM blueprint. Structurally, this was scaled down to a lightweight 80 million parameters matrix consisting of 6 hidden layers with a hidden vector dimension of 256 and an intermediate feed-forward network size of 1024. For text processing and context window mapping up to 256 tokens, the architecture splits processing tasks across 8 multi-query attention heads alongside 4 key-value heads to optimize memory travel.

The dataset of 500,000 arithmetic expressions was programmatically generated and structured inside a structured ChatML template utilizing standard `<|im_start|>` and `<|im_end|>` delimiters to explicitly separate the user query from the expected output. Each line (formatted) is structured as follows:

Listing 4.3: ChatML Template for LLM Arithmetic

```
1 <|im_start|>user
2 Calculate: 4+10//2*3
3 <|im_end|>
4 <|im_start|>assistant
5 19
6 <|im_end|>
```

Each mathematical expression obeys to three rules - only integers with value from 1 to 50 are allowed, the numbers of operands ranges from 2 to 4 and the division is the integer one.

Finally, optimization was launched from absolute scratch using the AdamW optimizer and standard half-precision (fp16) floating-point mathematics. This optimization process was sustained over 2,000 training steps at an aggressive learning rate of $5e-4$ and an expansive batch size of 128 sequences. The training took 13:37 minutes and generated a 166 MB model.

To assess the operational efficacy of the deployed model, three distinct empirical test trials were conducted natively on the endpoint hardware, with each batch evaluating 1,000 unique mathematical expressions. Results are shown in Table 3.

Pass rate and Execution time for a Qwen2ForCausalLM architecture model

Evaluation Phase	Total Test Cases	Correct Solves	Accuracy (%)	Avg. sec/prediction
Evaluation Batch 1	1,000	301	30.10%	0.1962
Evaluation Batch 2	1,000	324	32.40%	0.2277
Evaluation Batch 3	1,000	298	29.80%	0.1994
Aggregated Total	3,000	923	30.77%	0.2078

While an average operational latency of approximately 0.2 seconds per prediction demonstrates highly efficient local inference throughput, the corresponding 30.77% mean accuracy rate remains insufficient for reliable deployment.

In an attempt to fix the accuracy bottleneck, the network and training limits were scaled up: the hidden size was doubled from 256 to 512, an extra layer was added (moving from 6 to 7), and the attention heads were doubled to 16 query and 8 key-value heads. Additionally, the maximum context window was expanded to 512 tokens, and the training life cycle was lengthened from 2,000 to 3,000 steps to give the larger model more time to master the math logic. This magnified the number of parameters and the model size to 360 million and 334 MB, respectively. The training time was of 29:52. Table 4.

Pass rate and Execution time for a Qwen2ForCausalLM expanded architecture model

Evaluation Phase	Total Test Cases	Correct Solves	Accuracy (%)	Avg. sec/prediction
Evaluation Batch 1	1,000	382	38.20%	0.2586
Evaluation Batch 2	1,000	383	38.30%	0.3103
Evaluation Batch 3	1,000	381	38.10%	0.3395
Aggregated Total	3,000	1,146	38.20%	0.3028

Despite having simple arithmetic expressions and scaling up the model's layers and parameters, the modification yielded a very modest improvement, increasing the overall accuracy by only about 8% compared to the previous version. This limited progress suggests that simply enlarging the network size is not enough to effectively resolve the complex arithmetic logic constraints.

b) Building from Pretrained Models

Recognizing the cognitive limitations of small architectures trained from scratch, a more sophisticated transfer-learning strategy was adopted by fine-tuning a pre-trained, 3-billion-parameter base model (Qwen2.5-3B-Instruct) [70]. To execute this resource-intensive pipeline within a standard 16 GB VRAM budget, the base model was initialized in a highly compressed 4-bit precision format using the Unsloth framework, and Parameter-Efficient Fine-Tuning (PEFT) was applied using Low-Rank Adaptation (LoRA) adapters with a rank and alpha setting of 16 ($r=16$, $\alpha=16$) across all critical attention and feed-forward modules. This configuration allowed the training sequence to target only about 1% of the network's overall parameters, optimizing the model over a short 150-step schedule using a synthetic dataset of 250,000 algorithmic math expressions structured in ChatML format. Following structural alignment, the adapter weights were merged and compiled directly into a balanced q4_k_m GGUF quantization format. The training duration was of 38:47 minutes and the generated GGUF artifact occupies 1.8 GB in disk. The results presented in Table 5 are only better in accuracy than the previous attempts.

Pass rate and Execution time for Qwen2.5-3B-Instruct using Q4_K_M quantization

Evaluation Phase	Total Test Cases	Correct Solves	Accuracy (%)	Avg. sec/prediction
Evaluation Batch 1	250	124	49.60%	3.5301
Evaluation Batch 2	250	138	55.20%	3.2313
Evaluation Phase 3	250	115	46.00%	3.3559
Aggregated Total	750	377	50.27%	3.3724

In order to test each 250 expression batch, the inference model took 13.42 minutes on average. Reaching the 50% threshold is not much, as it means it is off target half of the times, and does not make up for the cost. Squeezing a 3-billion-parameter network onto the endpoint hardware expanded the binary file footprint to a heavy 1.8 gigabytes and inflated the inference latency to an unviable 3.37 seconds per simple calculation. This increase in execution time creates an unacceptable performance trade-off for a synthetic arithmetic dataset; simply put, demanding multiple seconds of intense CPU strain for a system that still fails half the time is functionally counterproductive for localized, real-time application, and much worse for a malware that wants to stay hidden. These empirical boundaries clearly prove that handling Large Language Models just for solving simple specialized tasks like calculating the final value of an expression is not an option.

In addition, from an engineering and software (malware) developing perspective, iterating on localized large language models introduces a prohibitive feedback loop that fundamentally stalls rapid prototyping. When

development cycles require hours of training and seconds of hardware strain per inference pass during verification loops, testing turns remarkably costly, slow, and counterproductive. This poses a direct barrier to the development of such techniques.

4.3 MLP as a Calculator

While substituting an entire Large Language Model for straightforward I/O software components is not a good approach given the test results in the previous section, the underlying paradigm introduces distinct obfuscation advantages. Specifically, executing logic encoded purely as floating-point weights requiring only a standard matrix multiplication runtime to operate. Such technique seems promising. By stripping away the structural bloat, autoregressive token loops, and attention mechanics inherent to LLMs, this architecture effectively collapses into a MLP (Multi-Layer Perceptron). This transition isolates the foundational pattern-matching capabilities of neural arrays within a highly efficient, lightweight feed-forward network, capturing the representational flexibility of deep learning without its associated framework dependencies and large compute penalties.

For a simple arithmetic calculator, an architecture consisting of 2 input nodes — representing the two operands of the operation — utilizes a hidden layer of 32 neurons to map the continuous numeric inputs into a higher-dimensional space for non-linear logic representation. This intermediate signals are subsequently compressed down to a single output neuron tasked with performing linear regression to resolve the final scalar sum, as previewd in Image 3.

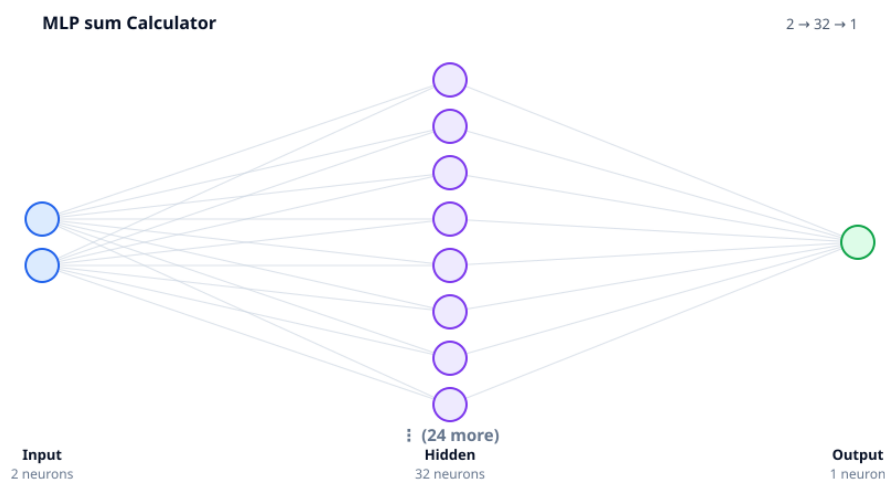


Figure 3: MLP as Sum architecture

Operating this 2-32-1 network topology only requires sequential matrix transformations suddenly the program

achieves microsecond-level execution as shown in Table 6. The aforementioned model took 4:45 minutes to train, on a 250000 entry dataset with sums of integers ranging from -10000 to 10000. The strategy was using PyTorch to train, but in contrast to earlier attempts, save the weights to a file and have the architecture as a template written in C. The weights data file has the size of 129 float point integers (4 bytes), 0.516 KB.

Pass rate and Execution time for sum calculator MLP

Evaluation Phase	Total Test Cases	Correct Solves	Accuracy (%)	Avg. μs/prediction
Evaluation Batch 1	1,000	829	82.90%	0.804
Evaluation Batch 2	1,000	826	82.60%	1.791
Evaluation Batch 3	1,000	836	83.60%	4.130
Aggregated Total	3,000	2,491	83.03%	2.242

For a fundamental operation such as the addition of two integers, yielding an 80% accuracy rate remains suboptimal. Nevertheless, this represents a major improvement over the Large Language Model baseline, marking an performance increase of over 30% in the pass rate, and dropping the execution time to the nanosecond order. Python v3.12.9 and the `angr` module version v9.2.222 were setup for a single solution find symbolic execution. It took, on average from three runs, 1.391 seconds to find a solution for a normal input output built-in sum function. Using the same script, for the version using the custom MLP as summation, the program did not find a solution, did not halt and crashed due to program explosion. This is due to the fact that every time the ReLU activation is activated the symbolic engine forks the execution into the two possible outcomes (zero or positive output). Therefore, doubling the trace space with each subsequent loop iteration drives an exponential expansion of the system's state space [5, 57]. The result proves this obfuscation technique introduces resistance against this analysis type. Intuitively, any program that inherits this multi-layered architecture will systematically hinder symbolic execution frameworks.

4.4 Domain Generation Algorithm

A DGA (Domain Generation Algorithm) is a programmatic technique used by malware authors to dynamically create a vast number of pseudo-random domain names. These generated domains serve as temporary communication channels for infected machines attempting to establish a connection with their command-and-control (C2) servers.

By utilizing a shared, time-dependent or sequence-dependent seed (such as the current system date), both the malware and the threat actor know exactly which domains will be generated at any given temporal point. Because

the algorithm can generate thousands of distinct domains while the attackers only need to register one or two to regain control, DGAs effectively bypass traditional security countermeasures like static IP blacklisting or domain reputation filtering [50].

The subsequent phase of the evaluation process involved attempting to model a Domain Generation Algorithm using deep learning techniques. From a functional input-output perspective, a DGA operates as a deterministic string-generation function, accepting a variable seed and mapping it to a pseudo-random, structurally valid hostname string value. To capture the sequential nature of alphanumeric generation, a Long Short-Term Memory (LSTM) recurrent neural network architecture was deployed; however, this approach yielded limited success.

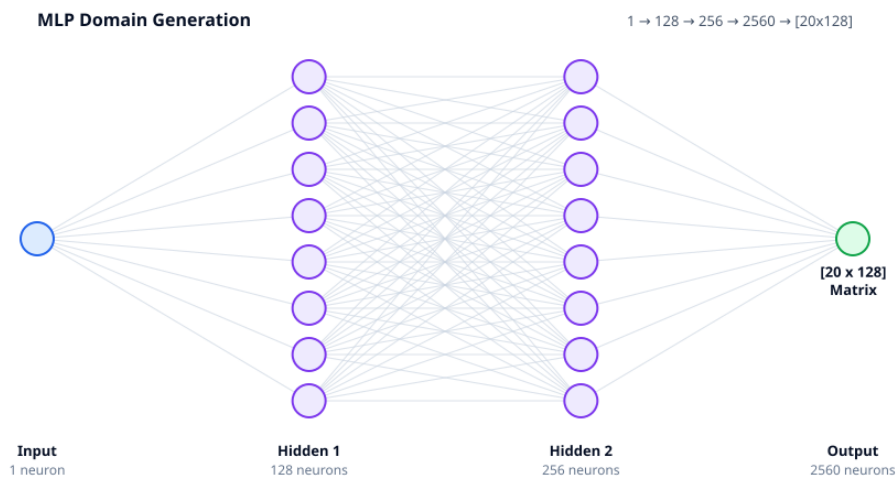


Figure 4: MLP as DGA architecture

Instead, the provided implementation imaged by Figure 4 outlines a non-recurrent, lookup-based embedding Multi-Layer Perceptron (MLP) engineered to map a specific discrete index directly to a static string representation. Upon receiving an index token, the network utilizes an embedding layer to map the index into a dense continuous vector space of size 128. This representation is passed through an intermediate projection layer that doubles the feature space dimension before applying a Rectified Linear Unit (ReLU) activation to introduce structural non-linearity. Finally, the network utilizes a massive linear output layer that maps the hidden features to a flattened tensor representing the total character probability space. This output is reshaped into a three-dimensional tensor, turning the classification task into a simultaneous, multi-position character estimation matrix where each index of the string length receives its own distinct probability distribution over the vocabulary. The generation of a valid string is described at Figure 5. In the diagram the size 20x128 is specified, representing the standard ASCII vocabulary size (128) and a length of 20 characters maximum for a hostname string.

The concrete architecture was implemented using PyTorch, with its final weights and execution logic exported

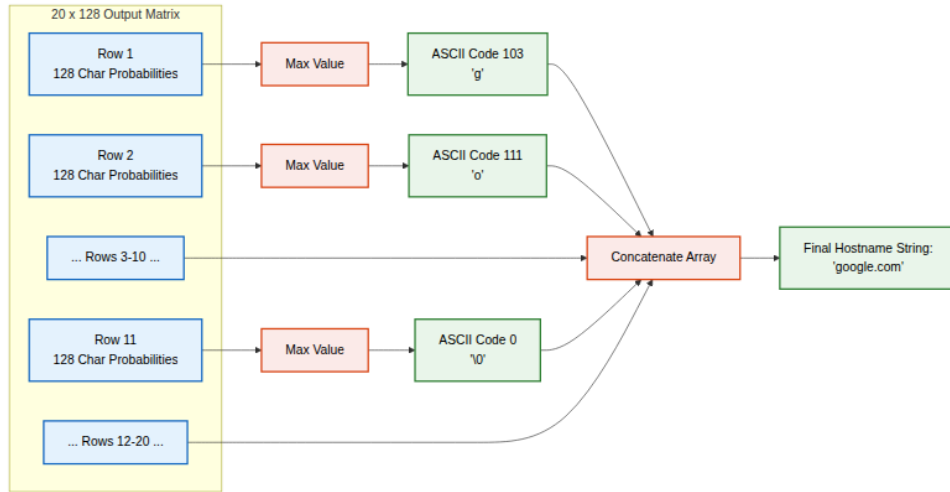


Figure 5: MLP DGA Selection Process

directly into C source templates, matching the deployment workflow established for the basic summation calculator network. The training dataset was constructed by extracting hostnames from the SecLists repository's discovery file, services-names.txt. Each base service name was modified by appending the static suffix "-micac" and sequentially combining it with ten arbitrarily selected top-level domains (TLDs) to form the final corpus of structured target strings, 14190 entries. The prediction matrix for this dataset was extended to 40x128. To achieve exact mapping between each discrete seed and its corresponding target string, the network was intentionally overfitted using a learning rate of 0.2 across 1500 training epochs. The total training duration was of 1:22 minutes, and the data weights 13 MB. For the pass rate, the domains were compared with the respective counterpart in the dataset using the seed. Table 7 displays the optimal outcome for this evaluation phase, achieving a flawless 100% pass rate. This confirms that the network successfully overfitted the training corpus as intended, establishing a perfect lookup mapping. Furthermore, the model demonstrated exceptional computational efficiency, maintaining execution latencies strictly within the millisecond range per prediction.

Pass rate and Execution time for DGA MLP

Evaluation Phase	Total Test Cases	Correct Solves	Accuracy (%)	Avg. msec/prediction
Evaluation Batch 1	1,000	1,000	100.00%	3.162
Evaluation Batch 2	1,000	1,000	100.00%	3.061
Evaluation Batch 3	1,000	1,000	100.00%	2.955
Aggregated Total	3,000	3,000	100.00%	3.059

Listing 4.4: Hostname validity regular expressions in C

```

1 // --- MUST MATCH THESE TWO PATTERNS ---
2 // Validates total length (1-253) and bans spaces
3 const char *res_global = "[a-zA-Z0-9.-]{1,253}$";
4 // Enforces 1-63 chars per label; bans empty labels
5 const char *res_labels = "^(\\.[^.]\\{1,63}\\|\\.)*\\.[^.]\\{1,63}$";
6
7 // --- MUST NOT MATCH THESE TWO PATTERNS ---
8 // Rejects labels starting or ending with a dash
9 const char *res_edges = "^-|-\\$|\\|\\.|-|\\.\\$";
10 // Rejects any label containing more than one dash
11 const char *res_double = "-[\\.]*-";

```

Although all domains generated within the valid seed space strictly adhered to the "service-micac26.tld" template and matched the training dataset perfectly, exploratory testing was conducted using seeds outside the training range to evaluate the model's behavior under out-of-bounds inputs. The subsequent 100000 out-of-range evaluations were validated using C hostname regular expressions to assess whether the network would still emit structurally well-formed, valid hostname strings or degenerate into unparseable character arrays.

The empirical data in Table 8 demonstrates a steep mathematical decay in structural validity as the input identifiers diverge from the bounded training seed space. Initially, near-boundary seeds maintain a fragile structural coherence (80.00% validity over 100 samples), but as the scale increases to 100,000 iterations, the accuracy plummets before ultimately triggering a critical runtime system crash. Because the network was intentionally overfitted using a non-recurrent Lookup MLP architecture, it functions essentially as a continuous mathematical memorization table rather than an algorithmic generator. When forced to infer outside its predefined range, the hidden layer projections yield chaotic, unconstrained activations.

Hostname Validity for Out-Of-Bounds Seed Spaces

Evaluation Phase	Total Test Cases	Valid Hostnames	Validity (%)	Operational Status
OOB Proximity Batch 1	100	80	80.00%	Operational
OOB Proximity Batch 2	1,000	418	41.80%	Operational
OOB Proximity Batch 3	10,000	1,895	18.95%	Operational
OOB Proximity Batch 4	100,000	N/A	N/A	System Crash

4.5 Neural Hash Map

The empirical results confirm that the Multi-Layer Perceptron does not function as an generative or algorithmic Domain Generation Algorithm (DGA) in the traditional sense; instead, it operates mathematically as a high-dimensional hash map. By minimizing empirical risk to zero across a fixed, finite input space, the network's hidden layers act as non-linear key-value pairs. This moves towards the field of Implicit Neural Representations for data and highly resembles the work researched in [32]. The trade-offs of this approach compared to classical discrete lookup mechanisms are systematically detailed in Table 9.

Comparison: Traditional Data Structures vs. Implicit Neural Data Representations

Architectural Metric	Traditional Hash Map / Array	Overfitted MLP Representation
Member Access Time Complexity	$O(1)$ Direct	$O(1)$ Single execution of Network
Storage Scaling Model	Linear Scaling ($O(n)$ with Elements)	Sub-linear
Retrieval Exactness	Deterministic 100% Bit Match	Lossy
State Modification Cost	Fast Value Mutation ($O(1)$)	Need to Retrain Model

The main advantage of using the neural data representation is its obfuscation value.

4.6 MLP as Packer

While leveraging INRs (Implicit Neural Representations) as continuous storage media is an established concept in coordinate-based vision and signal processing [44, 58], adapting this mechanism for binary asset packaging and programmatic data masking represents a novel architectural paradigm. Such idea was tested empirically and a neural packer using the same MLP network structure in the previous section was implemented based on the architecture schematized in Figure 6. At its core, it is still a DGA only with incremental integers starting at 1 as seeds. The dimensions of the matrix used for these tests were 20x128.

Below in Table 10 and Figure 7 the tool benchmarking results can be found for packing and extracting varied size files. Both the packing and extraction were performed in the end point hardware (Intel Core i7-9750H CPU (operating at 2.60GHz) and an NVIDIA GeForce GTX 1660 Ti mobile GPU (6 GB GDDR6 VRAM with CUDA 13.2 support). Only the packing phase was integrated with Compute Unified Device Architecture capabilities, for the extracting to simulate an average unaccelerated device. Time measurement was done through hyperfine 1.20.0, starting with one warmup run and three executions of each operation. Loss is the difference between a perfect

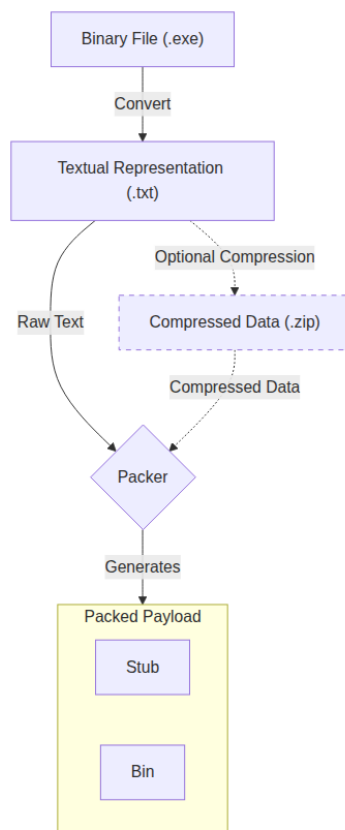


Figure 6: MLP as DGA architecture

decompression (100% accuracy) and the recreation accuracy obtained for each file.

Computational Runtime Scaling for Packing and Extraction Operations - NPicker

Target File	File Size (KiB)	Pack Time (s)	Extract Time (s)	Loss (%)	Weight Size (MiB)
file1	1.3	4.896	0.213	0.00%	2.7
file2	6.6	5.884	1.081	0.00%	3.0
file3	61.3	22.066	10.236	0.00%	5.7
file4	159.2	51.004	26.315	0.00%	10.6
file5	606.1	182.803	109.857	0.00%	32.9

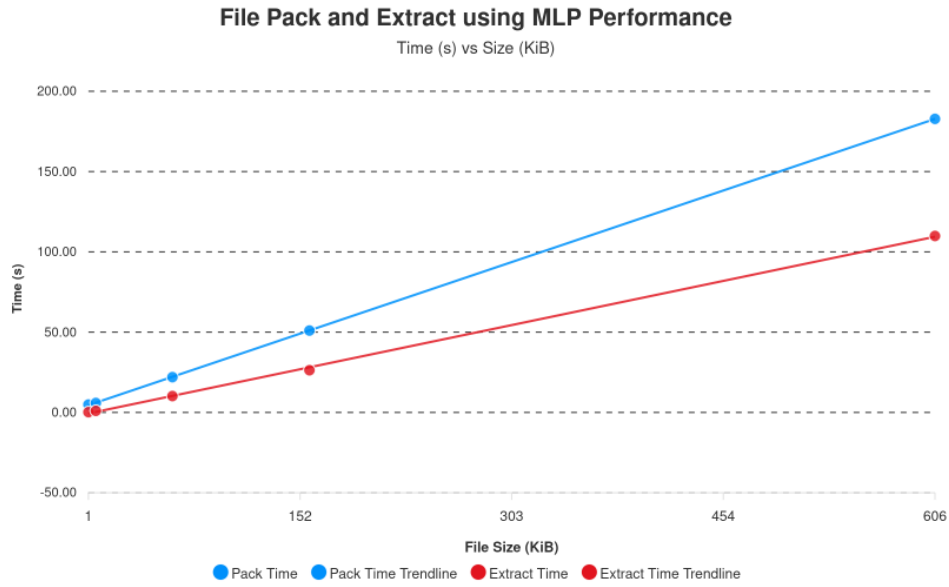


Figure 7: MLP Packer Performance

The pack time is approximated by the linear regression $y = 0.2947x + 4.1416$ where, y is the time and x the size. For extracting, the function $y = 0.1818x - 0.7977$ fits the data. Finally, for the disk space of the weight storage file the relationship falls under $y = 51.1066x + 2713.7167$, with the size increasing more than 50 times the original.

Through the lens of analysis, it is confirmed that the reconstruction pipeline operates with absolute bitwise fidelity, ensuring that all target binaries and assets emerge from the process entirely intact with zero loss. However, while functional integrity is maintained, runtime latency scales linearly with the spatial dimension of the input payload ($O(n)$ algorithmic complexity), introducing severe performance bottlenecks as file sizes increase. Proposing this

linear coefficient model onto a standard 2 MiB (2048 KiB) binary results in an estimated extraction window of approximately 6.2 minutes, presenting an exceptionally high execution overhead for practical deployment. Beyond processing latency, the framework exhibits a critical spatial paradox regarding its final physical footprint; according to the structural regression models, the compiled file size expands by an adverse factor of approximately $50\times$ relative to the baseline asset. This introduces a technical irony, as a tool architecturally categorized as a "packer" ultimately functions as an asset expander, shifting its viable utility away from storage optimization and placing it strictly as a high-latency obfuscation layer where spatial efficiency is entirely traded for opacity.

Nevertheless, in order to validate these findings, the empirical evaluation framework established in the Methodological Approach chapter is to be applied here. A live Conficker malware sample — packed with UPX [53] — was selected as the baseline input (*File 1*). It was unpacked using upx 5.1.1 and processed through the neural packing pipeline to generate the payload (*File 2*). These metrics are accessed in Table 11.

Structural and Complexity Comparison: Baseline UPX vs. Neurally Packed Payload

Metric	Control (<i>File 1</i>)	Custom (<i>File 2</i>)	Delta / Scaling
File Size (Bytes)	163,032	10,321,671	$63.3107\times$
Assembly Size (Lines)	64,984	3,290,068	$50.6289\times$
Cyclomatic Complexity	5,263	186,624	+181,361
Cosine Similarity	N/A	N/A	0.9193

According to the results, the custom obfuscated sample yields functional conservation (0.9193 Cosine Similarity) at the expense of massive structural distortion. Based on observations from the previous section the $63.31\times$ surge in raw byte volume is no surprise (expanding from 163 KB to over 10.3 MB) and a $50.63\times$ inflation in assembly lines. Concurrently, the binary's control-flow graph undergoes severe amplification, with cyclomatic complexity exploding by +181361 degrees as traditional instructions are mapped into dense neural network matrices. To evaluate the signature-evasion capabilities of the processed binaries, an initial multi-engine static scanning phase was conducted using VirusTotal 3.0 [35]. This allowed for an immediate comparative assessment of detection rates across commercial signature repositories. Figures 8 and 9 show the GUI yielding the resulting score.

Such evaluation from the AV group is plausible given that the Conficker malware family is already known and databases match its signature easily. For the custom module, five Anti Virus models detected it as malicious based

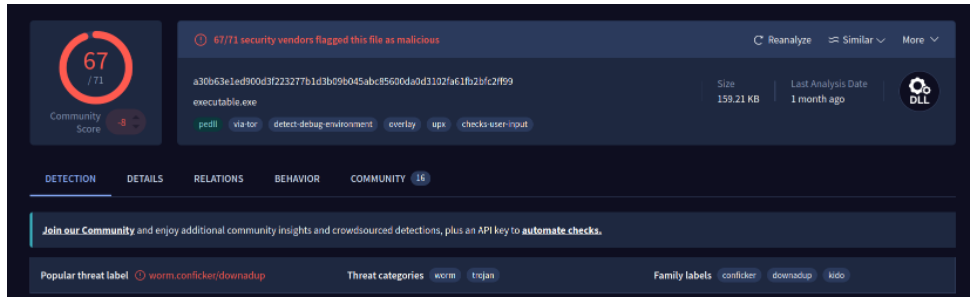


Figure 8: *File 1* VT score

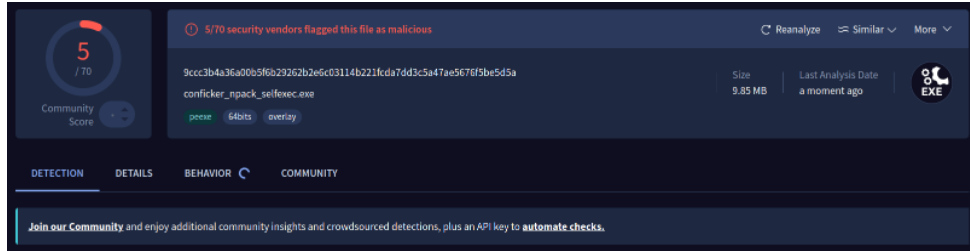


Figure 9: *File 2* VT score

on heuristic value methods and through machine learning based analysis as depicted in Figure 10.

Bkav Pro	W32.Malware.6386C1DA	CrowdStrike Falcon	Win/malicious_confidence_70% (D)
Elastic	Malicious (high Confidence)	SecureAge	Malicious
Symantec	ML.Attribute.HighConfidence	Acronis (Static ML)	Undetected

Figure 10: *File 2* AV Detections

For entropy measurement, both program were analyzed in Detect It Easy v3.21. The following table 12 contains this data, where the normal user-facing executable Notepad++ v8.9.6.2 was added for comparison. NPACKer stands for the implementation developed during research.

Both packed versions have high entropy which lead DIE's default packer detection through entropy heuristics. Interestingly, with detail provided in Image 11, it is possible to check that throughout the file the most repeated byte value by far is 0xBE, followed by 0x3F.

A consistent structural pattern is observed across other compiled payloads, such as the Cacheset64Wrapper.exe 9.6MiB packed executable discussed later in this work. As illustrated in fig. 12, the byte-entropy distribution highlights an identical high-density signature. This profile corroborates that the structural

UPX vs NPACKer Entropy comparison

File	Entropy
NotePad++	6.93850
Conficker packed with UPX	7.91192
Conficker packed with NPACKer	7.30085

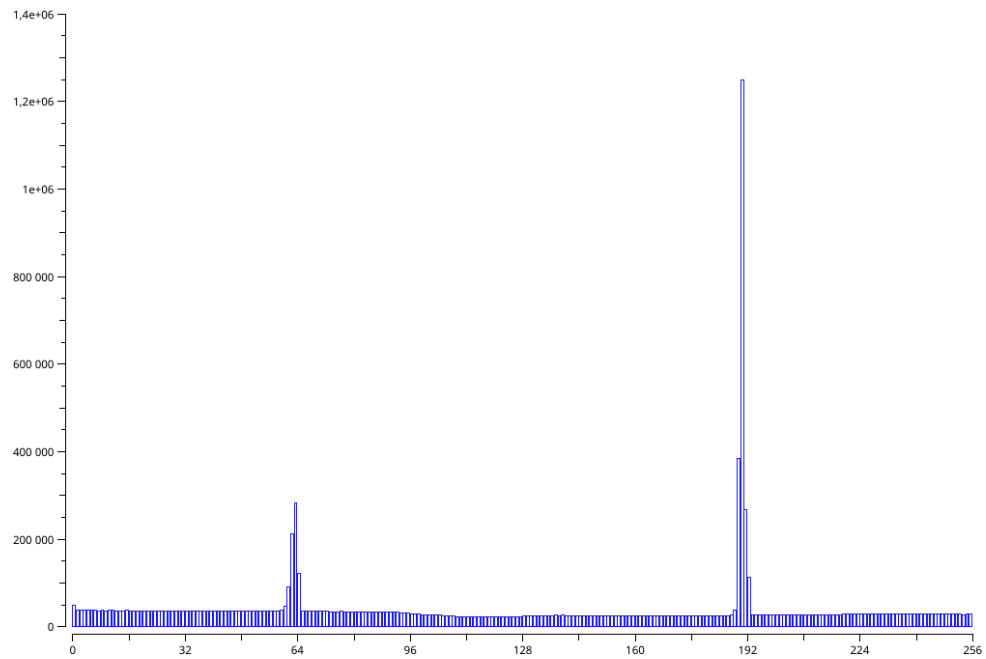


Figure 11: NPACKed Conficker Byte level Entropy

inflation and elevated entropy anomalies are architectural characteristics inherent to the packer, rather than isolated artifacts of a specific binary payload.

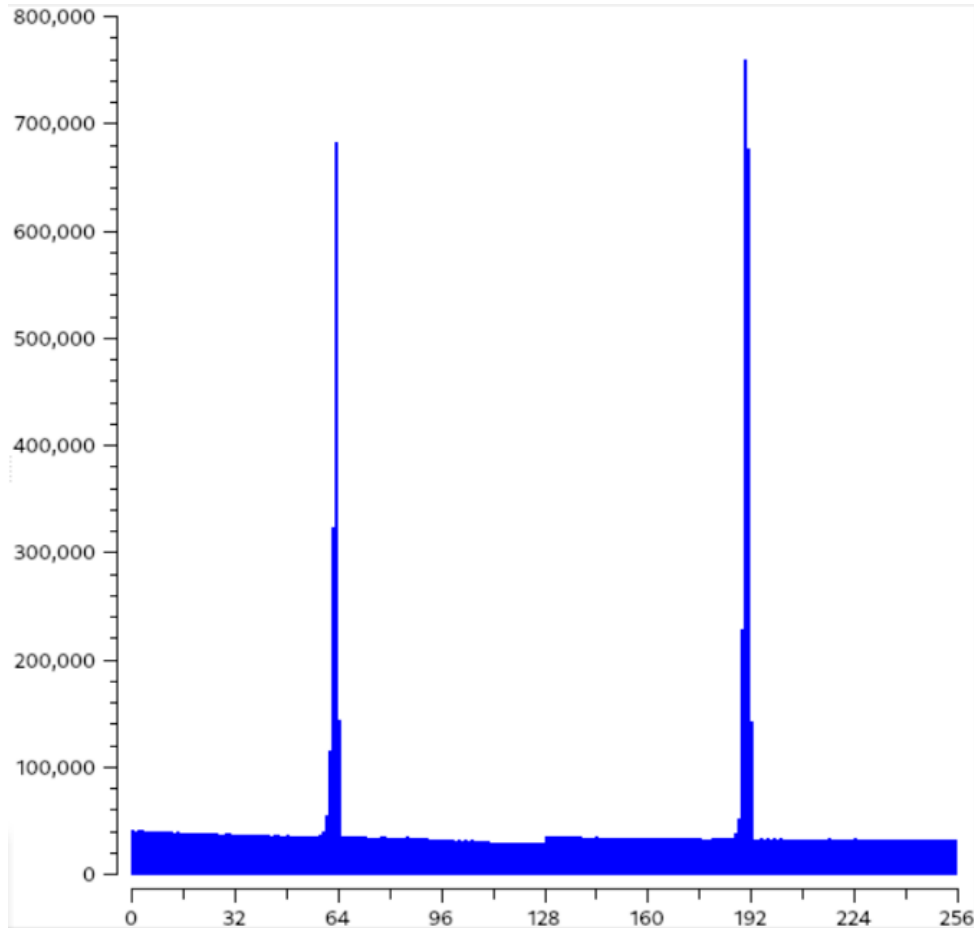


Figure 12: NPacked Cacheset64Wrapper Byte level Entropy

Following an exhaustive reconfiguration of the training parameters, which included doubling both the hidden dimension and vocabulary size, alongside expanding the maximum index length to 100 bytes, the resulting entropy profile maintained its relative structural amplitude while exhibiting a distinct visual translation to the left 13. Rather than destabilizing the binary layout, this persistent distribution behavior suggests that the shifting pattern acts as an architectural footprint of the packer's embedded neural data. Consequently, this unique structural signature provides a consistent, empirical baseline that can be used to reliably identify and classify binaries compiled under this specific MLP framework.

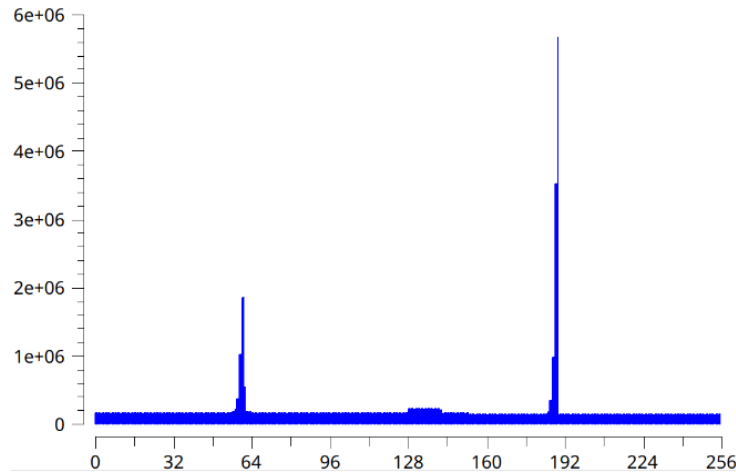


Figure 13: NPacked file Byte level Entropy an expanded Network

4.7 Transformer model as Packer

To build a more resilient compressor, advanced alternatives bypass basic multilayer perceptrons in favor of modern autoregressive sequence architectures. Tools like NNCP optimize context modeling by leveraging the Transformer framework. This family of networks replaces recurrent loops entirely with multi-head self-attention layers [6, 7, 64]. Below in Table 13 are the results for packing and extracting the same files as in the previous section, using the endpoint hardware without the CUDA api (only CPU).

Computational Runtime Scaling for Packing and Extraction Operations - NNCP

Target File	File Size (KiB)	Pack Time (s)	Extract Time (s)	Loss (%)	Weight Size (KiB)
file1	1.3	0.433	0.420	0.00%	1.4
file2	6.6	2.378	2.188	0.00%	6.8
file3	61.3	20.741	20.221	0.00%	57.5
file4	159.2	58.897	52.955	0.00%	156.4
file5	606.1	228.077	229.948	0.00%	549.8

While NNCP offers a significantly better compression ratio (reproducing archives at nearly a 1-to-1 level), it introduces performance trade-offs. Packing speeds remain comparable to the MLP-centric design, in contrast, the extraction times are noticeably longer. A linear regression was calculated as viewed in Figure 14, pinpointing that NNCP took twice the extracting time as the standard MLP ($y = 2.0970x - 0.8009$). For an agent deploying on a

target system, the two most critical metrics are the final weight size (which determines the physical disk footprint) and the extraction time (which directly impacts how long the executable takes to launch and initialize in memory). To accurately simulate a typical scenario endpoint lacking dedicated graphics hardware, the benchmark was once again executed strictly on the aforementioned CPU. Although a lower packing time is naturally preferred to simplify development and testing workflows, it is less relevant to the core evasive and operational objectives outlined in the methodological approach section.

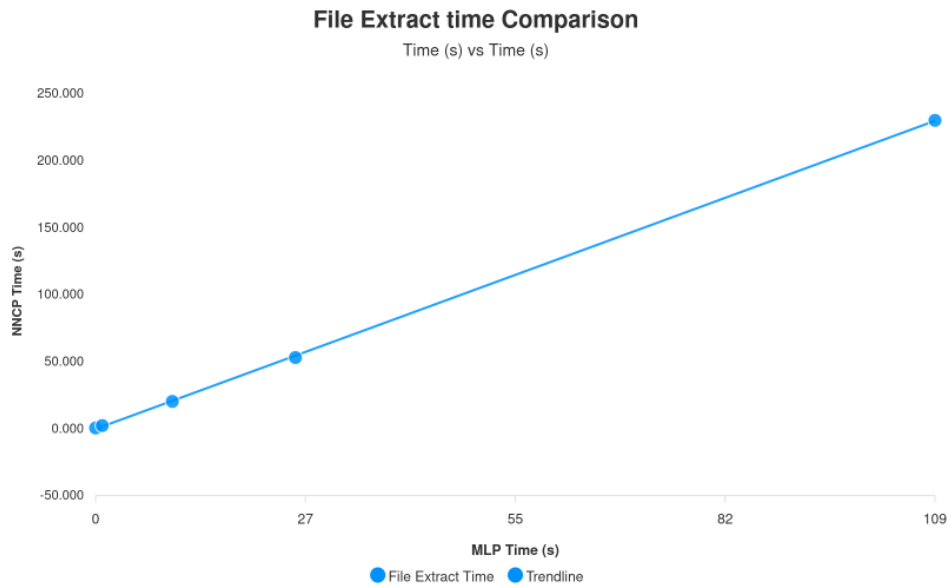


Figure 14: MLP vs NNCP Extracting Performance

4.8 Custom Remote Access Trojan

A Trojan horse is a type of malicious software disguised as legitimate, benign code that misleads users regarding its true intent in order to execute unauthorized actions, compromise data, or gain unauthorized access to a system [54]. Over the last 30 years RATs (Remote Access Trojans) have evolved from niche cybercriminal tools into highly standardized, ubiquitous commodities for global espionage and scams [63].

Following the individual development of the Domain Generation Algorithm and the neural packing framework, these components were integrated into a unified synthetic remote access utility to evaluate the cumulative efficacy of the combined architecture. This multi-layered implementation serves to determine whether the application of these components provides a measurable advantage in terms of evasion and stealth justifying the associated spatial and computational overhead.

While the MITRE ATT&CK framework catalogues prevalent Windows persistence vectors such as the widely recognized modification of registry Run keys (T1547.001) or DLL search order hijacking (T1574.001), these

techniques are heavily monitored by modern endpoint detection [51]. To establish persistence, the Silent Process Exit Monitoring mechanism was selected. Officially classified as a sub-technique of Image File Execution Options (IFEO) Injection (T1546.012), this administrative debug feature allows an arbitrary monitor application to execute seamlessly via Windows Error Reporting (WerFault.exe) whenever a designated target process terminates silently [45]. This key was configured to track the sysinternals suite application `Cacheset64.exe`, that demands admin privileges by default.

To create stealth, a process-hiding mechanism was implemented targeting the Windows Task Manager (`taskmgr.exe`). This is achieved by injecting a custom runtime library (DLL) into the target process memory space to perform API hooking on the native system call `NtQuerySystemInformation`. When the Task Manager queries the `SystemProcessInformation` class to enumerate active threads and binaries, the hook intercepts the returned doubly linked list of `SYSTEM_PROCESS_INFORMATION` structures, surgically modifying the `NextEntryOffset` fields to unlink and bypass the tracking agent's PID [19].

To test the network communication layer, the remote access malware was configured to use a standard TCP connection that looks up its server via normal DNS queries [28]. The malware built uses the DGA with the neural obfuscation technique described earlier.

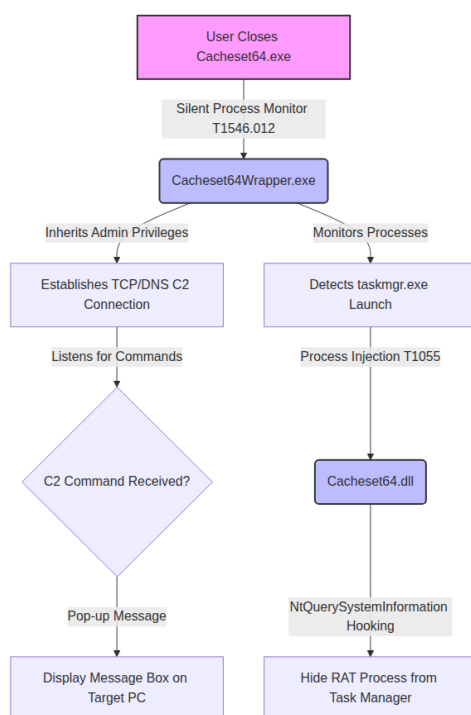


Figure 15: simple RAT architecture

The implemented test sample brings these previously detailed mechanisms together into a single, cohesive

execution chain as illustrated in Figure 15. The sequence initiates when the termination of `Cacheset64.exe` triggers the Silent Process Monitor infrastructure (T1546.012), forcing the operating system to automatically execute `Cacheset64Wrapper.exe` with inherited, high-privilege administrative context. Once active, this wrapper initializes the TCP outbound channel to establish contact with the remote command-and-control server and await instructions. Simultaneously, the wrapper runs a localized background routine to track systemic process creation telemetry; the moment the launch of `taskmgr.exe` is observed, the wrapper handles the necessary process injection (T1055) to deploy `Cacheset64.dll`, which immediately enforces the `NtQuerySystemInformation` API hooks to mask the primary wrapper's active background footprint from the user interface.

To package these components for execution, all binaries were bundled into a single installer using Inno Setup 6.2.2 running compiled in Wine 10.0 on Kali Linux 2026.1. When executed on the target system, this setup package automatically extracts the required files and configures the necessary registry keys for the Silent Process Monitor before launching any user-facing applications. The sample makes use of the neural packing mechanisms developed both for payload 1 and payload 2. The structural composition of the binary was evaluated via entropy analysis using the Detect It Easy framework, as illustrated in 16. The analysis reveals an elevated global entropy footprint across the file. This high-density profile is primarily driven by the massive volume of embedded neural network parameters residing within the remaining sections.

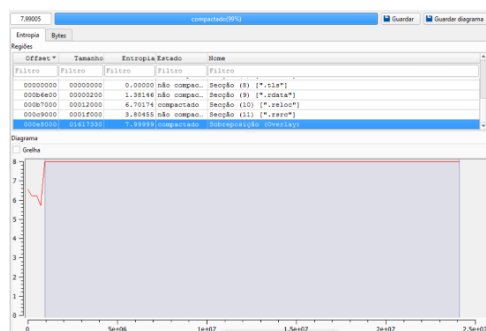


Figure 16: simple RAT Entropy per Section

To evaluate the operational behavior and dynamic indicators of compromise (IOCs) of the generated binary, the custom malware sample was detonated within a local Windows 10.0.10240 enterprise emulated with qemu 11.0.0. Furthermore, to verify inbound and outbound orchestration telemetry and ensure that the runtime engine safely initiates connection sequences with the designated command-and-control (C2) server, network emulation was conducted via the Mandiant Flare FakeNet-NG framework [31].

The deployment pipeline successfully installed the Trojan in 48.79 minutes. As illustrated in fig. 17, the payload

successfully initiated a connection sequence with the designated command-and-control (C2) server, confirming that the obfuscated architecture preserves core operational functionality.

```

8:55 PM DNS Server: Received A request for domain 'dis.msfnccsl.com' from lschost.exe (1896)
8:55 PM DNS Server: Received A request for domain 'dis.msfnccsl.com' from lschost.exe (1896)
8:41 PM Divertor: Cacheset64wupper.exe (292) requested TCP 192.168.122.187/4444
8:55 PM Divertor: Cacheset64wupper.exe (292) requested TCP 192.168.122.187/4444
8:55 PM DNS Server: Received A request for domain 'fed.update.microsoft.com' from lschost.exe (1896)

```

Figure 17: Connection to C2 by custom RAT

For baseline evaluation, a control sample was compiled by deploying the malware using standard techniques replacing the developed ones throughout the paper. Both the baseline executable and the modified variant were subsequently submitted to VirusTotal for comparative detection analysis. The resulting detection rates and security vendor telemetry are detailed below in Figures 18.

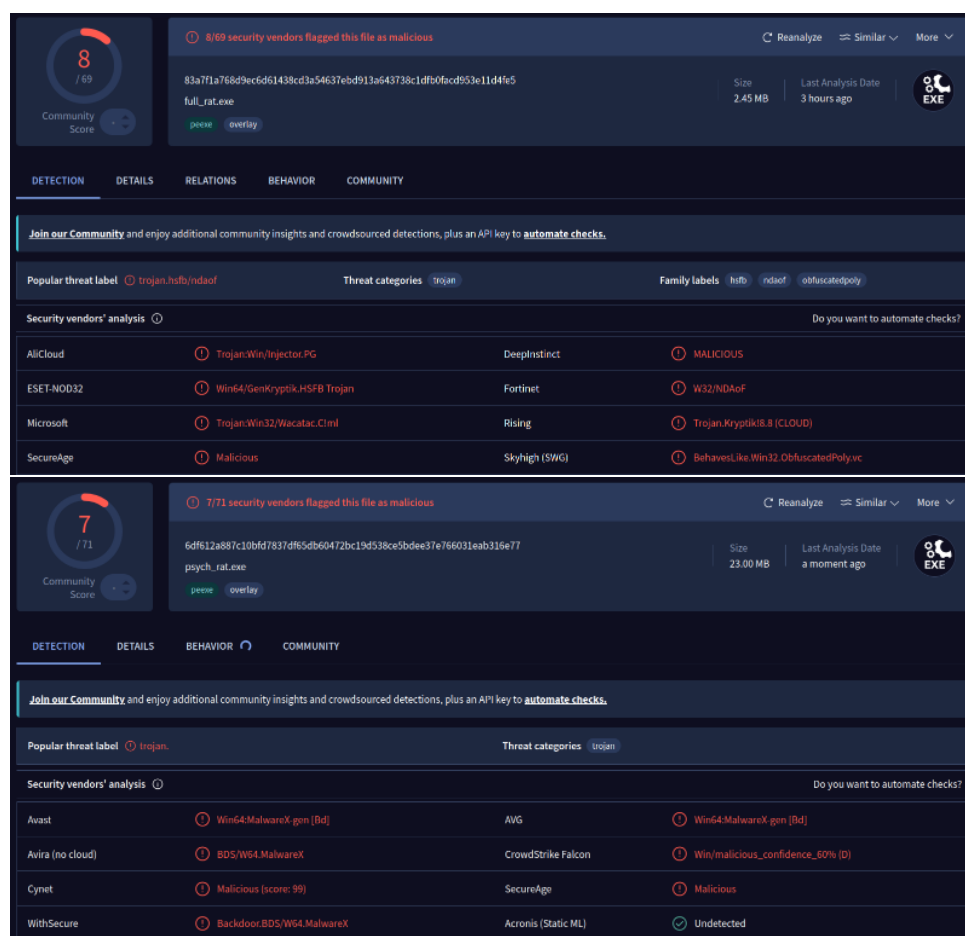


Figure 18: Control (top) vs Custom (bottom) Sample analyzed by VT 3.0

Comparative analysis on VirusTotal demonstrates that the developed obfuscation technique successfully alters the binary's static signature profile despite substantial payload inflation. The standard control sample (full_rat.exe, 2.45 MB) yielded an 8/69 detection rate, triggering specific, well-defined threat signatures associated with the Kryptik, Wacatac, and Injector families. In contrast, the custom sample employing the neural packing layers (psych_rat.exe, 23.00 MB) reduced the absolute detection rate to 7/71. Crucially, the custom technique stripped away these specific, identifiable family classifications, shifting the security vendor telemetry entirely toward generic, heuristic-based alerts such as MalwareX-gen and confidence-scored markers, which proves a slight improvement over the normal version.

5 Conclusion

In conclusion, Large Language Models (LLMs) remain too cumbersome for practical malware deployment due to their heavy architectural dependencies and a systemic lack of hardware acceleration on standard end-user environments. While LLMs excel at conceptual induction, they lack the deterministic precision required for strict code obfuscation, rendering the prospect of executing even a compact 0.5B parameter model silently in the background highly impractical. A more viable alternative lies in leveraging smaller, specialized neural networks in a combined fashion to execute discrete operational tasks. Beyond merely serving as opaque predicates or dynamic code obfuscators, Implicit Neural Representations establish a resilient barrier against well-established detection and deobfuscation tools; by mapping execution logic into an expansive array of numerical weights, they exploit the reality that raw binary data by itself carries little to no semantic information for static analyzers to decode. The empirical test results gathered throughout this study demonstrate that while exploring INRs and neural networks for input/output (I/O) reproduction leaves a vast landscape of potential yet to be uncovered, while current state-of-the-art commercial packers and standard evasion methods remain entirely sufficient to perform a more than satisfiable code obfuscation.

Bibliography

- B. Abrath, B. Coppens, S. Volckaert, and B. De Sutter. Obfuscating windows dlls. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 24–30, 2015. 10.1109/SPRO.2015.13.
- S. Ahn, S. Ahn, H. Koo, and Y. Paek. Practical binary code similarity detection with bert-based transferable similarity learning. In *Proceedings of the 38th Annual Computer Security Applications Conference, ACSAC '22*, page 361–374, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450397599. 10.1145/3564625.3567975. URL <https://doi.org/10.1145/3564625.3567975>.
- E. Alkhateeb, A. Ghorbani, and A. Habibi Lashkari. Identifying malware packers through multilayer feature engineering in static analysis. *Information*, 15:1–26, 02 2024. 10.3390/info15020102.
- V. Attias, N. Bellec, G. Menguy, S. Bardin, and J.-Y. Marion. Augmenting search-based program synthesis with local inference rules to improve black-box deobfuscation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS '25*, page 4529–4543, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715259. 10.1145/3719027.3765134. URL <https://doi.org/10.1145/3719027.3765134>.
- R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018. ISSN 0360-0300. 10.1145/3182657. URL <https://doi.org/10.1145/3182657>.
- F. Bellard. Lossless data compression with neural networks. Technical report, bellard.org, may 2019. URL <https://bellard.org/nncp/nncp.pdf>. Accessed: 2026-06-24.
- F. Bellard. Lossless data compression with neural networks. Technical report, bellard.org, dec 2021. URL https://bellard.org/nncp/nncp_v2.1.pdf. Accessed: 2026-06-24.
- M. Brengel and C. Rossow. YARIX: Scalable YARA-based malware intelligence. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3541–3558. USENIX Association, Aug. 2021. ISBN 978-1-939133-24-3. URL <https://www.usenix.org/conference/usenixsecurity21/presentation/brengel>.
- K. Brezinski and K. Ferens. Metamorphic malware and obfuscation: A survey of techniques, variants, and generation kits. *Security and Communication Networks*, 2023(1):8227751, 2023. <https://doi.org/10.1155/2023/8227751>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1155/2023/8227751>.
- B. Casey, K. Damian, A. Cotaj, and J. C. S. Santos. An empirical study of safetensors’ usage trends and developers’ perceptions, 2025. URL <https://arxiv.org/abs/2501.02170>.
- O. Chaplia, H. Klym, and K. Babii. A comparative study of inference frameworks for Node.JS microservices on edge devices. *Advances in Cyber-Physical Systems*, 10(2):113–118, 2025. 10.23939/acps2025

.02.233. URL <https://science.lpnu.ua/sites/default/files/journal-paper/2025/nov/41049/vse-113-118.pdf>.

T. Colangelo. Digital rights management in video games: their impact on performance and the legal structure connected to them. Bachelor's thesis, Haaga-Helia University of Applied Sciences, 2020. Available at: <https://www.theseus.fi/handle/10024/345056>.

C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 184–196, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919793. 10.1145/268946.268962. URL <https://doi.org/10.1145/268946.268962>.

M. Dalla Preda. *Code obfuscation and malware detection by abstract interpretation*. Doctoral dissertation, Università degli Studi di Verona, Verona, Italy, December 2007. URL <https://hdl.handle.net/20.500.14242/113956>. Settore INF/01 - Informatica. Course: Informatica (Cycle 19).

B. De Sutter, S. Schrittwieser, B. Coppens, and P. Kochberger. Evaluation methodologies in software protection research. *ACM Comput. Surv.*, 57(4), Dec. 2024. ISSN 0360-0300. 10.1145/3702314. URL <https://doi.org/10.1145/3702314>.

E. Debas, N. Alhumam, and K. Riad. Unveiling the dynamic landscape of malware sandboxing: A comprehensive review. *Preprints*, December 2023. 10.20944/preprints202312.1009.v1. URL <https://doi.org/10.20944/preprints202312.1009.v1>.

M. Di Gennaro, M. D'Onghia, M. Polino, S. Zanero, and M. Carminati. *PackHero: A Scalable Graph-Based Approach for Efficient Packer Identification*, page 253–274. Springer Nature Switzerland, 2025. ISBN 9783031976230. 10.1007/978-3-031-97623-0_16. URL http://dx.doi.org/10.1007/978-3-031-97623-0_16.

W. Elersy, A. Feizollah, and N. Anuar. The rise of obfuscated android malware and impacts on detection methods. *PeerJ Computer Science*, 8:e907, 03 2022. 10.7717/peerj-cs.907.

S. Eresheim, R. Luh, and S. Schrittwieser. The evolution of process hiding techniques in malware - current threats and possible countermeasures. *Journal of Information Processing*, 25:866–874, 2017. 10.2197/ipsjip.25.866.

J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. S. Gaur, and M. Conti. A robust dynamic analysis system preventing sandbox detection by android malware. SIN '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334532. 10.1145/2799979.2800004. URL <https://doi.org/10.1145/2799979.2800004>.

D. Gibert, C. Mateu, and J. Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*,

153:102526, 2020. ISSN 1084-8045. <https://doi.org/10.1016/j.jnca.2019.102526>. URL <https://www.sciencedirect.com/science/article/pii/S1084804519303868>.

L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonyamy, and M. Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, ASIA CCS '20*, page 694–707, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367509. 10.1145/3320269.3384745. URL <https://doi.org/10.1145/3320269.3384745>.

J. Gobbo, F. Sarro, and P. Falcarin. The Ghidra metrics toolkit for obfuscated native code. In *Proceedings of the Joint National Conference on Cybersecurity (ITASEC & SERICS 2026)*, volume 4198 of *CEUR Workshop Proceedings*. CEUR-WS.org, April 2026. URL <https://ceur-ws.org/Vol-4198/paper58.pdf>.

N. Grech and Y. Smaragdakis. P/taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017. 10.1145/3133926. URL <https://doi.org/10.1145/3133926>.

W. Holder, J. T. McDonald, and T. R. Anel. Evaluating optimal phase ordering in obfuscation executives. *SSPREW-7*, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353878. 10.1145/3151137.3151140. URL <https://doi.org/10.1145/3151137.3151140>.

W. Hua, T. Wong, F. Sun, L. Pan, A. Jardine, and W. Y. Wang. InductionBench: LLMs fail in the simplest complexity class. In W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26526–26546, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. 10.18653/v1/2025.acl-long.1287. URL <https://aclanthology.org/2025.acl-long.1287/>.

G.-C. Huang and T.-H. Lai. Legacy code, live risk: Empirical evidence of malware detection gaps. *Applied Sciences*, 15(22), 2025. ISSN 2076-3417. 10.3390/app152211862. URL <https://www.mdpi.com/2076-3417/15/22/11862>.

D. Jeremiah, H. Rafiq, V. T. Ta, M. Usman, M. Raza, and M. Awais. Niom-dga: Nature-inspired optimised ml-based model for dga detection. *Computers Security*, 157:104561, 2025. ISSN 0167-4048. <https://doi.org/10.1016/j.cose.2025.104561>. URL <https://www.sciencedirect.com/science/article/pii/S0167404825002500>.

A. Karnik, S. Goswami, and R. Guha. Detecting obfuscated viruses using cosine similarity analysis. In *First Asia International Conference on Modelling Simulation (AMS'07)*, pages 165–170, 2007. 10.1109/AMS.2007.31.

J. Karthik, P. P. Amritha, and M. Sethumadhavan. Video game drm: Analysis and paradigm solution. In *2020*

11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), pages 1–4, 2020. 10.1109/ICCCNT49239.2020.9225560.

F. Khorrami, R. Karri, and P. Krishnamurthy. Real-time multi-modal subcomponent-level measurements for trustworthy system monitoring and malware detection, 2025. URL <https://arxiv.org/abs/2501.13081>.

T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. 10.1145/3183713.3196909. URL <https://doi.org/10.1145/3183713.3196909>.

A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes. In *NDSS 2021, Network and Distributed Systems Security Symposium*, Virtual, United States, Feb. 2021. ISOC, Internet Society. 10.14722/ndss.2021.24475. URL <https://hal.science/hal-04611612>.

U. Kurt. Which quantization should i use? a unified evaluation of llama.cpp quantization on llama-3.1-8b-instruct, 2026. URL <https://arxiv.org/abs/2601.14277>.

C. Leka, C. Ntantogian, S. Karagiannis, E. Magkos, and V. S. Verykios. A comparative analysis of virustotal and desktop antivirus detection capabilities. In *2022 13th International Conference on Information, Intelligence, Systems Applications (IISA)*, pages 1–6, 2022. 10.1109/IISA56318.2022.9904382.

B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu. MBA-Blast: Unveiling and simplifying mixed Boolean-Arithmetic obfuscation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1701–1718. USENIX Association, Aug. 2021. ISBN 978-1-939133-24-3. URL <https://www.usenix.org/conference/usenixsecurity21/presentation/liu-binbin>.

C. Luo and X. Wang. Latency–throughput tradeoffs of onnx runtime, tensorrt-llm, vllm, and triton: An empirical comparison on 1b–3b parameter llm inference. *Spectrum of Research*, 6(1), Apr. 2026. URL <https://spectrumofresearch.com/index.php/sr/article/view/49>.

R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security Privacy*, 5(2):40–45, 2007. 10.1109/MSP.2007.48.

T. László and A. Kiss. Obfuscating c++ programs via control flow flattening. In *Obfuscating C++ Programs via Control Flow Flattening*, volume 30, 06 2007.

M. Maghanaki, S. Keramati, F. F. Chen, and M. Shahin. Generation of a multi-class iot malware dataset for cybersecurity. *Electronics*, 14(21), 2025. ISSN 2079-9292. 10.3390/electronics14214196. URL <https://www.mdpi.com/2079-9292/14/21/4196>.

- K. Malakhov. Deploying llms on cpu-only environments with llama.cpp library set: Medlocalgpt project case. 10 2025.
- E. Masabo, K. Kaawaase, J. Otim, and D. Hanyurwimfura. Structural feature engineering approach for detecting polymorphic malware. pages 716–721, 11 2017. 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.125.
- A. Mercado, J. Villalobos, and E. Meneses. Performance evaluation of deep learning formats: A comparative study of onnx and pytorch for inference efficiency and portability. In *2025 IEEE 7th International Conference on BioInspired Processing (BIP)*, pages 1–6, 2025. 10.1109/BIP68491.2025.11489132.
- B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020. URL <https://arxiv.org/abs/2003.08934>.
- O. Moe. Persistence using globalflags in image file execution options – hidden from autoruns.exe, apr 2018. URL <https://oddvar.moe/2018/04/10/persistence-using-globalflags-in-image-file-execution-options-hidden-from-autoruns-exe/>. Accessed: 2026-06-24.
- A. Mohseni, M. H. Moaiyeri, A. Amirany, and M. Hadi Rezayati. Protecting the intellectual property of binary deep neural networks with efficient spintronic-based hardware obfuscation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(7):3146–3156, 2024. 10.1109/TCSI.2024.3397925.
- S. Mohseni, S. Mohammadi, D. Tilwani, Y. Saxena, G. K. Ndawula, S. Vema, E. Raff, and M. Gaur. Can llms obfuscate code? a systematic analysis of large language models into assembly code obfuscation, 2025. URL <https://arxiv.org/abs/2412.16135>.
- T. Muralidharan, A. Cohen, N. Gerson, and N. Nissim. File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. *ACM Comput. Surv.*, 55(5), Dec. 2022. ISSN 0360-0300. 10.1145/3530810. URL <https://doi.org/10.1145/3530810>.
- C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851, 2021. 10.1109/SP40001.2021.00012.
- D. Plohmann, K. Yakdan, M. Klatt, J. Bader, and E. Gerhards-Padilla. A comprehensive measurement study of domain generating malware. 08 2016.
- K. Rahal, A. Riahi, and T. Debatty. Dataset of apt persistence techniques on windows platforms mapped to the mitre attck framework. In *2025 28th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, pages 17–24, 2025. 10.1109/ICIN64016.2025.10943025.

- A. Raj, S. K. G, H. K. Chaubey, and S. Arelli. Llm-based model compression for efficient on-device learning. In *2025 IEEE 5th International Conference on Human-Machine Systems (ICHMS)*, pages 251–256, 2025. 10.1109/ICHMS65439.2025.11154251.
- J. Ramamoorthy, N. K Shashidhar, and C. Varol. Packers and features: Efficacy of static analysis for packed linux malware. In *2025 13th International Symposium on Digital Forensics and Security (ISDFS)*, pages 1–6, 2025. 10.1109/ISDFS65363.2025.11012083.
- S. Rashid, S. Baker, O. Alsaif, and A. Ahmad. Detecting remote access trojan (rat) attacks based on different lan analysis methods. *Engineering, Technology Applied Science Research*, 14:17294–17301, 10 2024. 10.48084/etasr.8422.
- M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi. Technical report: Hardening code obfuscation against automated attacks, 2022. URL <https://arxiv.org/abs/2106.08913>.
- S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1), Apr. 2016. ISSN 0360-0300. 10.1145/2886012. URL <https://doi.org/10.1145/2886012>.
- S. Shiqi, W. Meng, and S. Prateek. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.
- V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein. Implicit neural representations with periodic activation functions, 2020. URL <https://arxiv.org/abs/2006.09661>.
- W. Song. a framework for automated similarity analysis of malware. Master’s thesis, Concordia University, September 2014. URL <https://spectrum.library.concordia.ca/id/eprint/978935/>. Unpublished.
- V. Tejedor, C. Zuheros, C. Peláez-González, D. Herrera-Poyatos, Andrés, A. Herrera-Poyatos, and F. Herrera. The H-Elena Trojan Virus to Infect Model Weights: A Wake-Up Call on the Security Risks of Malicious Fine-Tuning. *arXiv preprint arXiv:2504.03823*, 2025. URL <https://arxiv.org/abs/2504.03823>.
- theZoo. thezoo: A repository of live malwares for your own joy. <https://github.com/ytisf/theZoo>, 2026. Accessed: 2026-06-25.
- L. D. Tomasi, C. D. Sipio, A. D. Marco, and P. T. Nguyen. Simplicity by obfuscation: Evaluating llm-driven code transformation with semantic elasticity, 2025. URL <https://arxiv.org/abs/2504.14024>.

- V. Valeros and S. Garcia. Growth and commoditization of remote access trojans. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*, pages 454–462, 2020. 10.1109/EuroSPW51379.2020.00067.
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023. URL <https://arxiv.org/abs/1706.03762>.
- N. Veeranna and B. C. Schafer. Efficient behavioral intellectual properties source code obfuscation for high-level synthesis. In *2017 18th IEEE Latin American Test Symposium (LATS)*, pages 1–6, 2017. 10.1109/LATW.2017.7906738.
- VX Underground. Vx underground: The largest collection of malware source code, samples, and papers on the internet. <https://vx-underground.org/>, 2026. Accessed: 2026-06-25.
- C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *2001 International Conference on Dependable Systems and Networks*, pages 193–202, 2001. 10.1109/DSN.2001.941405.
- J. Woodbridge, H. S. Anderson, A. Ahuja, and D. Grant. Predicting domain generation algorithms with long short-term memory networks, 2016. URL <https://arxiv.org/abs/1611.00791>.
- F. Xiang, D. Gong, C. Yang, and F. Liu. Enhanced branch obfuscation based on exception handling and encrypted mapping table. In *Proceedings of the ACM Turing Celebration Conference - China, ACM TURC '19*, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371582. 10.1145/3321408.3326668. URL <https://doi.org/10.1145/3321408.3326668>.
- A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang, G. Dong, H. Wei, H. Lin, J. Tang, J. Wang, J. Yang, J. Tu, J. Zhang, J. Ma, J. Yang, J. Xu, J. Zhou, J. Bai, J. He, J. Lin, K. Dang, K. Lu, K. Chen, K. Yang, M. Li, M. Xue, N. Ni, P. Zhang, P. Wang, R. Peng, R. Men, R. Gao, R. Lin, S. Wang, S. Bai, S. Tan, T. Zhu, T. Li, T. Liu, W. Ge, X. Deng, X. Zhou, X. Ren, X. Zhang, X. Wei, X. Ren, X. Liu, Y. Fan, Y. Yao, Y. Zhang, Y. Wan, Y. Chu, Y. Liu, Z. Cui, Z. Zhang, Z. Guo, and Z. Fan. Qwen2 technical report, 2024. URL <https://arxiv.org/abs/2407.10671>.
- G. Yüce, G. Ortiz-Jiménez, B. Besbinar, and P. Frossard. A structured dictionary perspective on implicit neural representations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19228–19238, June 2022.
- S. Zhang, J. Zhao, Q. Yu, C. Xia, Z. Wang, X. Feng, and H. Cui. The new compiler stack: a survey on the synergy of llms and compilers. *CCF Transactions on High Performance Computing*, 8(2):148–179, Jan. 2026. ISSN 2524-4930. 10.1007/s42514-025-00270-x. URL <http://dx.doi.org/10.1007/s42514-025-00270-x>.

T. Ziebermayr. Protecting intellectual property rights of industrial software. *Procedia Computer Science*, 180:862–866, 2021. ISSN 1877-0509. <https://doi.org/10.1016/j.procs.2021.01.336>. URL <https://www.sciencedirect.com/science/article/pii/S1877050921003902>. Proceedings of the 2nd International Conference on Industry 4.0 and Smart Manufacturing (ISM 2020).

L. Zobernig, S. D. Galbraith, and G. Russello. When are opaque predicates useful? In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 168–175, 2019. 10.1109/TrustCom/BigDataSE.2019.00031.