A Formal Grammar Approach to Metamorphic Binary Transformation

Matteo Ciccaglione, Pierciro Caliandro, and Alessandro Pellegrini

Tor Vergata University of Rome, Rome, Italy
{matteo.ciccaglione,pierciro.caliandro,alessandro.pellegrini}@uniroma2.it

Abstract. We present a declarative reformulation of metamorphic binary transformation using translational attribute grammars. Rather than introducing new transformation families, we reformulate well-known metamorphic techniques (instruction substitution, register remapping, expansion and reduction) as grammar productions with associated semantic attributes, yielding composability, stochastic diversity, and formally checkable semantic constraints. We demonstrate the feasibility of this approach through a metamorphic engine for x86-64 ELF binaries and evaluate it against signature scanners (*VirusTotal*), image-based Convolutional Neural Networks (CNNs), and feature-guided tree-based models. Using `MetaMe` as a baseline, our grammar-based engine achieves higher evasion rates across all evaluated classifiers. We further show that grammar-generated variants are effective for *adversarial training*, enabling detection models to learn invariant features that resist structural evasion. The declarative structure of the grammar also enables a rule-by-rule validation methodology, allowing individual transformations to be isolated and verified via bounded translation validation.

Keywords: Metamorphism · Obfuscation · Generative Grammars

1 Introduction

Obfuscation is a widely adopted technique across many areas of computer science, ranging from detection evasion [7, 25] to intellectual property protection [9, 42]. A particularly interesting class of obfuscation is *dynamic obfuscation*, which relies on a program's ability to modify itself, potentially during execution. In this way, the program rewrites or regenerates parts of its own code during execution, even multiple times. Prominent examples include *mimicry* (*mimimorphism*), *polymorphism*, and *metamorphism* [40].

Metamorphism can be seen as the art of automatically rewriting an application by changing its code and structure while preserving its semantics. It works by replacing code fragments with semantically equivalent instruction sequences, possibly reordering blocks, inserting unnecessary or redundant instructions, and performing register or operand substitutions, so that each new variant is syntactically different but behaviourally indistinguishable from the original program.

When metamorphism is applied to malware, it becomes a concrete threat, since this family of transformations is particularly effective at evading signature-based and machine-learning (ML) powered detection techniques [26]. The continuous production of new variants from a single original version makes it hard for classical pattern-based approaches to maintain reliable signatures. This has pushed researchers towards more sophisticated, semantics-aware and adversarial machine learning techniques [11, 5, 4], where metamorphic engines are used to generate challenging datasets for training and evaluation.

The individual building blocks of metamorphism (instruction substitution, register remapping, expansion and reduction of instruction sequences) have been present in advanced engines for decades. Early tools such as MetaPHOR and W32.Zmist already implemented recursive expansion/reduction strategies and multi-stage mutation pipelines in a procedural fashion [37]. The core limitation of these approaches is the lack of a rigorous formal specification: because transformations are expressed as static, procedurally embedded substitution tables, there is no unified formal model governing their interactions, correctness guarantees remain informal, and the resulting binaries may exhibit unnatural syntactic structures that aid detection [8].

Our Contribution. We propose a *declarative* reformulation of metamorphic transformations using a *translational attribute grammar*, shifting the paradigm from procedurally embedded substitution tables to syntax-driven generation. The transformations we encode are well known; the novelty lies in their *formal specification*: representing transformation rules as grammar productions with associated semantic attributes yields composability, stochastic diversity, and semantic guarantees that procedural engines cannot provide. We operate at the disassembly level, using `Capstone` to map the binary to a mnemonic representation and `Keystone` to reassemble the result. The grammar's three-level hierarchy (syntactic, semantic, and translational) provides structured abstraction within the instruction domain, without recovering higher-level source code (e.g., C). Attribute constraints reject any transformation branch that violates operand-type or data-flow invariants, avoiding the unnatural syntactic structures that aid detection. The formal specification yields three concrete benefits demonstrated in our evaluation: (1) because each transformation is expressed as an independent grammar production, individual rules can be isolated and verified via bounded translation validation (Section 4.1), a methodology that is not available in procedural engines where transformations are entangled in monolithic code; (2) the `revert` mechanism enforces semantic constraints at generation time, ensuring that invalid transformation branches are pruned before they produce incorrect output; and (3) stochastic diversity arises as a natural consequence of grammar ambiguity and non-deterministic rule selection, rather than being an ad-hoc mechanism layered onto a procedural pipeline.

We implement the framework through Tahr [13], which also enables defining a transformation context carried consistently throughout the process, supporting complex transformations such as using free memory areas as temporary variables.

Our comparison baseline is `MetaMe` [28], one of the few publicly available open-source metamorphic engines for Linux ELF binaries already used in the literature. It applies substitutions only when the replacement has the same byte length as the original, which limits its transformation space and makes the comparison conservative in our favour [1, 15, 22].

We structure our evaluation around three research questions.

- **RQ1** (Feasibility): can attribute grammars serve as a practical specification mechanism for metamorphic transformation, producing semantically correct and structurally diverse variants?
- **RQ2** (Evasion): do grammar-generated variants evade signature-based and ML-based malware detectors, and how does their evasion capability compare with `MetaMe`?
- **RQ3** (Defensive utility): can grammar-generated adversarial samples be used to harden detection models via adversarial training?

The evaluation confirms a positive answer to all three. The remainder of the paper is structured as follows: Section 2 defines the threat model. Section 3 describes the grammar-guided transformation. In Section 4 we present the implementation and experimental evaluation, while in Section 5 we discuss mitigations. Related work is reviewed in Section 6.

2 Threat Model

We consider a *full-blind attacker* threat model, which we classify as a *black-box, zero-knowledge evasion attack* in the taxonomy of problem-space adversarial attacks formalised by [29]. The adversary controls the binary prior to distribution but has no prior knowledge of the detection techniques deployed by the target: no information about the classifier architecture, feature extraction pipeline, training data, or decision boundary is available to the attacker. The attacker's objective is to evade malware analysis pipelines without altering the payload semantics. Because the transformations are applied entirely in the problem space (i.e., directly on binary instructions rather than on extracted feature vectors), the resulting variants are realisable executables by construction and do not require a differentiable path to the detector. This zero-knowledge assumption ensures that the applied transformations are broadly applicable and robust: since the attacker optimises for no specific classifier, evasion gains generalise across heterogeneous detection stacks rather than overfitting to a single model.

On the defensive side, we assume a post-incident forensics environment that deploys both traditional signature scanners and computationally intensive ML-based structural analysers, specifically CNN image-similarity models and feature-guided tree ensembles. Our evaluation reflects these two perspectives in distinct research questions. In RQ2, we measure the evasion rate of transformed binaries against detectors that have never observed metamorphic variants, directly instantiating the blind-attacker scenario described above. In RQ3, we shift to the *defender's* perspective and assess whether augmenting the training set with

grammar-generated variants (adversarial training) can harden the same detectors against future metamorphic evasion. These two settings are complementary and do not conflict: adversarial training is a defensive countermeasure in which the analyst, not the attacker, generates variants from training-set samples to improve model robustness. It does not grant the attacker any additional knowledge, nor does it relax the zero-knowledge assumption that governs the attack model.

3 Formalizing Transformation via Attribute Grammars

Metamorphism applies random perturbations to confuse signature-based detection while preserving program semantics. Attribute Grammars (AGs) provide the mechanism to encapsulate semantic rules directly in the transformation logic: by shifting transformation selection to a formal language processing task, we rely on *context-sensitive constraints* evaluated on instruction operands and types to ensure that alterations do not compromise correctness.

The Tahr framework [13] ingests AG specifications to synthesise a Transformation Engine (TE) orchestrating code manipulation across a threefold hierarchy: the *syntactic level* governs formal structure, the *semantic level* assigns meaning through attribute logic, and the *translational level* maps source constructs to target representations. Level 1 recognises individual mnemonic instructions from the disassembly text; level 2 categorises instructions by syntactic form; level 3 abstracts them to a fully semantic representation, independent of the underlying architecture. Once lifted to level 3, any symbol can be translated into one or more semantically equivalent level-3 sequences, or specialised back to a level-2 syntactic form, according to the rules of the grammar.

Tahr resolves rule ambiguity stochastically, selecting among applicable productions at random. This enables the exploration of highly diverse mutations. When a production branch fails to satisfy embedded semantic constraints, the `revert` directive aborts it and forces an alternative derivation, ensuring that even high-entropy transformations maintain strict semantic equivalence. The TE outputs a functionally identical binary with a fully novel internal organisation. The grammar's layered design separates the high-level transformation intent from the concrete instruction encoding. While porting to a new target ISA (e.g., ARM or RISC-V) would also require adapting architecture-specific semantic attributes and transformation rules, the modular structure of the grammar simplifies this effort compared with monolithic procedural engines.

To illustrate the actions of the TE, let us consider the mutation of the example instruction `movq %rsi,0x8(%rdi)`, by relying on the grammar specified in Listing 1. The parsing phase maps the instruction to the level-2 symbol mov_{rm}, which denotes a register-to-memory transfer—the attributes allow us to retain the instruction's specific operands. The symbol mov_{rm} is then lifted into the level-3 symbol $write_{data}$, which expresses the abstract concept of *data writing*. Thanks to attribute manipulation, $write_{data}$ is parameterized with attributed for the source (`%rsi`), base (`%rdi`), and offset (`0x8`). Every time the TE picks a transformation rule stochastically, it remains strictly bound by these context attributes. For

```
write_data -> save_data, lea_m_r,
↪  write_data, restore_data{
    if ($$.mem != 1 ||
    ↪  involves_rsp($$.base,$$.src)){
        revert;
    }
    $1.reg = $$.base;
    $2.base = $$.base;
    $2.offset = $$.offset;
    $2.dest = $$.base;
    $3.src = $$.src;
    $3.base = $$.base;
    $3.offset = 0;
    $3.mem = 1;
    $4.reg = $$.base;
    $4.mem = 0;
} | mov_r_m {
    if ($$.mem != 1){
        revert;
    }
    $1.base = $$.base;
    $1.src = $$.src;
    $1.offset = $$.offset;
} | save_data,restore_data{
    if (involves_rsp($$.base,$$.src)){
        revert;
    }
    $1.reg = $$.src;
    $2.dest = $$.base;
    $2.offset = $$.offset;
    $2.mem = 1;
};

save_data -> push_r{
    $1.reg = $$.reg;
};

restore_data -> pop_r{
    if ($$.mem != 0){
        revert;
    }
    $1.dest = $$.base;
} | pop_m{
    if ($$.mem != 1){
        revert;
    }
    $1.base = $$.base;
    $1.offset = $$.offset;
};
lea_m_r -> mov_r_r,add_i_r{
    $1.src = $$.base;
    $1.dest = $$.dest;
    $2.imm = $$.offset;
    $2.dest = $$.dest;
};
```

Listing 1: Example of Generative Rules

example, attempting to specialise this memory operation into a register-to-register transfer (a mov_{rr} level-2 symbol) would violate the destination type constraint. This transformation would trigger execution of the `revert` statement, which would abort the transformation, pruning the invalid branch in the generation tree and forcing the engine to select an alternative path.

Eventually, a valid expansion path is chosen. The first iteration of the `grow` algorithm [14] may expand the $write_{data}$ symbol into a more complex sequence of operations. As shown in the *Intermediate* column of Figure 1, $write_{data}$ could be specialized into a lea_{mr} level-2 symbol (to handle address calculation) and a sequence of level-3 symbols: $save_{data}$, to push the value onto the stack, and $restore_{data}$, to pop the value to the new destination. This changed sequence of operations could effectively obfuscate the data flow in the original application.

Then, the TE recursively operates on these three symbols: lea_{mr} could be lowered to a pair of level-1 symbols mov_{rr} and add_{ir}, while $save_{data}$ and $restore_{data}$ can be concretised into $push_r$ and pop_r, respectively. This recursive process continues until a defined maximum depth is reached. At that point, the original instruction `movq %rsi,0x8(%rdi)` is replaced by the final, semantically equivalent, sequence of operations, as shown in the *Output* column of Figure 1.

The example illustrates a key property (and a potential drawback) of pure expansion: applying expansion rules alone may lead to exponential growth in the executable over time. To counter this, we also encode reduction rules in our Tahr grammar. Reduction operates on an output buffer of generated symbols: Tahr

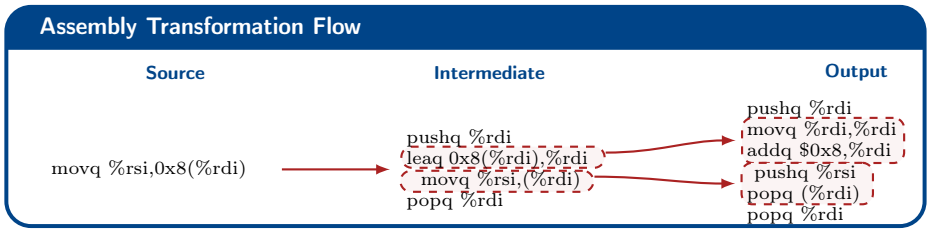


Fig. 1: Example Recursive Expansion

```
lea_m_r <- mov_r_r,add_i_r{
    if ($1.dest != $2.dest){
        revert;
    }
    $$.base = $1.src;
    $$.dest = $2.dest;
    $$.offset = $2.imm;
};
mov_r_m <- push_r,pop_m{
    $$.src = $1.reg;
    $$.base = $2.base;
    $$.offset = $2.offset;
};
mov_r_m <- lea_m_r,mov_r_m{
    if (!($1.base == $2.base && $2.offset
    ↪  == 0)){
        revert;
    }
    $$.base = $1.base;
    $$.src = $2.src;
    $$.offset = $1.offset;
};
```

Listing 2: Example of Reduction Rules

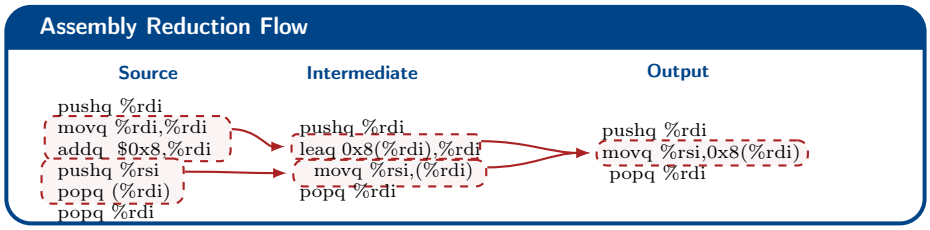
continuously evaluates the tail of the symbol stream and triggers reduction rules whose constraints are satisfied, performing in-place substitutions. To illustrate this shrinking capability, let us consider the *Source* column of Figure 2a, processed with the rules in Listing 2.

As shown in Figure 2a, Tahr stochastically combines mov_{rr} and add_{ir} into a lea_{mr}, and reduces $pop_m/push_r$ into a mov_{rm}. A recursive second pass can further consolidate mov_{rr} with lea_{mr}, though a residual pop_r symbol may remain, producing a final binary that differs from the original. By selectively applying reduction rules, the engine can retain specific intermediate expansions, mixing high-level abstractions with low-level encoding in the same output.

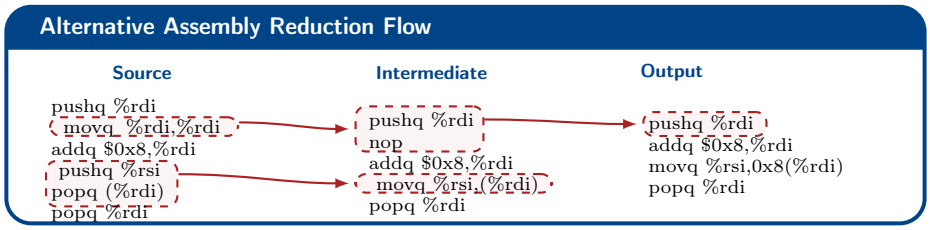
Alternative rule sets produce qualitatively different reductions. With the rules in Listing 3 (Figure 2b), attribute context identifies the first mov_{rr} as a null operation, reducing it to *nop*; a second pass then consolidates $push_r/nop$ into a single $push_r$—a form of *dead code elimination*. Rules from both listings can coexist in a single grammar, and their stochastic interplay ensures that metamorphic variants differ in both length and instruction composition at every generation.

4 Experimental Assessment

We developed a fully functional metamorphic engine targeting ELF binaries on x86-64, using `libelf` for parsing, `Capstone` [10] for disassembly, `Keystone` [21]



(a) Example Recursive Reduction using Rules in Listing 2



(b) Example Recursive Reduction using Rules in Listing 3

Fig. 2: Comparison of Recursive Reduction Strategies.

for reassembly, and `Tahr` with a custom attribute grammar covering a subset of the x86 instruction set. The full grammar and engine source are publicly available on Zenodo[1]. The grammar encodes both expansion and reduction rules across several instruction families: register-to-register and register-to-memory data transfers, immediate-to-register and immediate-to-memory operations, arithmetic and logical operations (`add`, `sub`, `and`, `or`, `xor`), `LEA`-based address computations, and stack operations (`push`, `pop`). The complete set of rules is listed in Listing 4; detailed attribute operations are omitted for brevity.

All transformations operate exclusively at the instruction level and do not alter the control-flow graph (CFG) of the binary. Conditional and unconditional branches are preserved as-is, and no basic-block reordering, CFG restructuring, or control-flow flattening is performed. In other words, the grammar targets the data-flow and instruction-encoding dimensions of the binary while leaving its control-flow structure intact. We note that the evasion performance of the engine is naturally dependent on the number and diversity of the rules encoded in the grammar: a richer grammar produces more diverse variants and spans a larger transformation space. The Zenodo artifact contains the complete grammar specification for reproducibility.

To address the three RQs stated in the introduction, we structure the evaluation as follows. We first validate the correctness of individual transformation rules and analyse computational cost (**RQ1**); we then measure evasion rates against signature-based and ML-based detectors (**RQ2**); and finally we assess

[1] https://doi.org/10.5281/zenodo.18467108.

```
lea_m_r <- mov_r_r,add_i_r{
    if ($1.dest != $2.dest){
        revert;
    }
    $$.base = $1.src;
    $$.dest = $2.dest;
    $$.offset = $2.imm;
};
nop <- mov_r_r{
    if ($1.src!=$1.dest){
        revert;
    }
};
push_r <- push_r,nop{
    $$.reg = $1.reg;
};
mov_r_m <- push_r,pop_m{
    $$.src = $1.reg;
    $$.base = $2.base;
    $$.offset = $2.offset;
};
mov_r_m <- lea_m_r,mov_r_m{
    if (!($1.base == $2.base && $2.offset
    ↪  == 0)){
        revert;
    }
    $$.base = $1.base;
    $$.src = $2.src;
    $$.offset = $1.offset;
};
```

Listing 3: Alternative Example of Reduction Rules

```
write -> i_mov_r_m{}
| save,restore{}
| i_mov_r_r{};

load -> i_mov_m_r {}
| i_lea_m_r,i_mov_m_r {}
| i_nop {};

save -> i_push_r {}
| i_sub_i_r,write {};

restore -> i_pop_r {}
| load,i_add_i_r {}
| i_pop_m {};

cond_jmp -> i_cmp_r_r,
↪  jump_be {}
| i_cmp_r_r,jump_ab {}
| i_cmp_r_r,jump_be {}
| i_cmp_r_r,jump_ae {};

jump_be -> i_jb {}
| i_jbu {};

jump_ab -> i_ja {}
| i_jg {};

jump_ae -> i_jae_u {}
| i_jae {};

jump_be -> i_jle {}
| i_jbe {};

i_jmp -> jump_ae, jump_be
↪  {}
| i_jz, i_jnz {}
| I_JMP;

i_sub_i_r -> i_add_i_r,
↪  i_sub_i_r {}
| I_SUB_I_R;
i_lea_m_r -> i_mov_r_r,
↪  i_add_i_r {}
| I_LEA_M_R;
```

Listing 4: Generative Rules adopted in our Case Study

whether grammar-generated variants can be used for adversarial training (**RQ3**, discussed in Section 5).

Reference Dataset. Our dataset is drawn from the VirusShare [32] 2019–2020 corpus (53,979 Linux samples; 12,443 targeting Intel), from which we uniformly sampled 4,000 malicious Linux ELF binaries targeting Intel x86-64. For the ML evaluation, we add 4,000 benign applications from GNU `binutils`, yielding a balanced 8,000-sample dataset split 70/15/15 into training, validation, and test partitions. The comparison baseline throughout is `MetaMe` [28], an engine widely adopted in the literature for evaluating detection robustness and generating adversarial training data [1, 15, 22].

4.1 Validating Transformation Rules

Guaranteeing semantic correctness of arbitrary binary transformations in general is an undecidable problem, and we do not claim that the grammar provides full correctness proofs for all possible inputs. Instead, we demonstrate that the

declarative structure of the grammar enables a principled *rule-by-rule validation methodology*: because each transformation is expressed as an independent grammar production, individual rules can be selectively enabled or disabled and verified in isolation against the original binary. This is a structural advantage over procedural engines, where transformations are entangled in monolithic code and cannot be easily isolated for independent testing.

To demonstrate this methodology, we adopt bounded translation validation using symbolic execution and SMT solving (`Z3` [27], `Alive2` [23]), treating the output of each rule as if it were the result of a custom compiler optimisation pass, i.e., the same methodology used to validate LLVM optimisation passes. We use `ReOPT` [34], which lifts an executable into an LLVM intermediate representation (IR), allowing `Alive2` to prove equivalence between the original and the mutated code.

Due to the *path explosion* problem inherent in symbolic execution, we constrained the verification scope to a minimal executable: a simple implementation of the coreutils' `wc` (word count) utility, complex enough to trigger most grammar rules while remaining tractable for symbolic execution tools. The verification proceeded as follows:

1. The source code was compiled to establish a baseline `wc` binary;
2. This baseline was processed through the metamorphic engine with specific rule subsets enabled, generating `wc_1`;
3. A second pass on `wc_1` triggered reduction rules, producing `wc_2`;
4. `ReOPT` lifted all three artifacts to LLVM IR; `Alive2` verified semantic equivalence of `wc_1` and `wc_2` relative to the original.

Following this approach, we validated all register-to-register and register-to-memory transfer rules, arithmetic rewrites, and LEA-based address-computation expansions. Transformations involving register-based `push`/`pop` instructions and conditional branches could not be verified with this toolchain: `Reopt` internally uses `Macaw` [33] to reconstruct the Control Flow Graph, which fails when the stack layout is altered or when conditional jumps are present without prior knowledge of the involved flags. This is a constraint of the current binary lifting tools; the excluded rules follow the same attribute-constraint design as the verified ones and are candidates for validation as these tools mature.

Beyond rule-level formal verification, we conducted a practical validation campaign on the full dataset: all 4,000 malware binaries were successfully transformed and reassembled by the engine, producing executable outputs in every case. When an instruction is not covered by any grammar production, the engine leaves it unchanged in the output stream, which guarantees that the resulting binary is always structurally valid and executable regardless of the input's instruction mix. This practical validation complements the formal methodology described above. The formal approach verifies the semantic correctness of individual rules in isolation: the verified subset (four rule groups out of ten in Listing 4, covering register-to-register and register-to-memory transfers, arithmetic rewrites, and LEA-based expansions) accounts for approximately one third

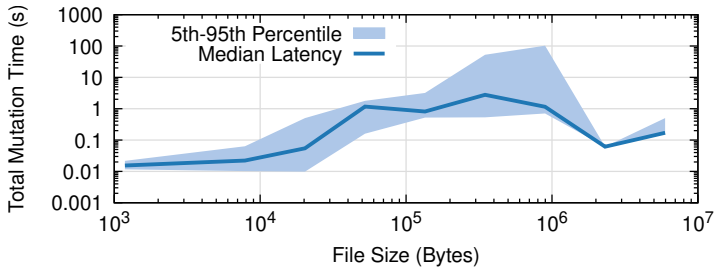


Fig. 3: Execution time of generation and reduction phases when the size of the binary increases.

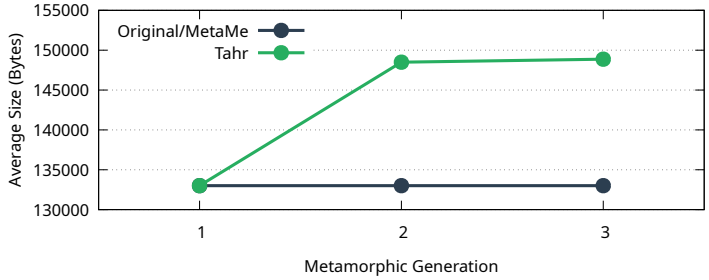


Fig. 4: Average variation in malware size across metamorphic generations.

of the grammar's productions, while the excluded rules (push/pop sequences and conditional branch rewrites) comprise the remaining two thirds. The successful transformation of every sample in the dataset demonstrates that the engine, including the unverified rules, operates reliably on real-world binaries and that the attribute-constraint mechanism prevents ill-formed outputs even for rules not yet amenable to symbolic verification.

4.2 Space and Time Overhead

Operational stealth requires that the mutation process not significantly delay payload execution; if transformation time exceeds the typical execution time of a malware sample, it could delay the payload, trigger timeouts, or raise user suspicion. We profiled the Tahr-driven TE across both expansion and reduction phases for increasing binary sizes (Figure 3). The results show no correlation between execution time and file size: cost depends on the density of matchable instructions and the non-deterministic rule selection rather than on binary size. Consequently, the transformation time is driven by the complexity of randomly selected mutation paths, not by file complexity. This is an important result, as it confirms that grammar-based metamorphism is practically viable and that recursive transformations can be bounded without sacrificing stealth.

We also assess space overhead by tracking binary size across metamorphic generations (Figure 4). In Figure 4, generation 0 denotes the original, unmodified binary; each subsequent generation corresponds to one complete pass of the metamorphic engine, comprising expansion followed by reduction. Figure 4 reports the average binary size at each generation. This aspect is important because metamorphic engines often bloat executables; a size anomaly alone could trigger detection.

Reference Dataset. We use the reference dataset described at the beginning of this section. Fine-tuning pre-trained CNNs and tree-based models rather than training from scratch provides sufficient statistical power while mitigating overfitting.

As shown in Figure 4, `MetaMe` variants do not change in size, as its same-size substitution constraint prevents bloat but restricts its transformation space. Our Tahr-based variants grow in generation one, as expansion rules fire extensively in the first pass and reduction rules are unlikely to apply before the engine has seen sequences that match its reduction patterns. When transitioning to generation two, the average malware size remains mostly stable, as reduction rules now come into play and counteract expansion, keeping the size growth bounded. Crucially, the moderate size increase in generation one is accompanied by a strictly larger transformation space and greater structural diversity. This is an important result: it shows that our grammar-based approach does not trade off size for diversity, but rather controls binary bloat through formal reduction rules while still supporting richer mutations than same-length substitution strategies.

4.3 Assessing Adversarial Evasion Capabilities

To evaluate the evasion and obfuscation capabilities of the approach, we use our reference dataset and `MetaMe` as the baseline. Our metric is the *evasion rate*—the percentage of samples that successfully bypass detection after one metamorphic transformation step. Note that the comparison with `MetaMe` is conservative in our favour: `MetaMe`'s same-size constraint limits its transformation space, so any improvement over `MetaMe` is a lower bound on what a more expressive engine can achieve against a detector that has not been specifically hardened against it.

We acknowledge that part of the observed evasion improvement is attributable to the richer set of transformations that our engine supports (variable-length rewrites, recursive expansion and reduction), which `MetaMe`'s same-size substitution constraint cannot express. However, this distinction does not diminish the contribution: the attribute grammar formulation is precisely what *enables* these richer transformations. Variable-length rewrites and recursive expansion require context-sensitive constraints to ensure semantic correctness, constraints that are naturally expressed as attribute conditions in the grammar but would require ad-hoc, error-prone bookkeeping in a procedural engine. The comparison therefore isolates a practical benefit of the formalism: the ability to safely express and compose a broader class of transformations. An ablation restricting our engine to same-size substitutions only would test whether the grammar formulation improves evasion within `MetaMe`'s transformation class, but such a comparison

Table 1: CNN Performance Against Metamorphic Attacks

CNN Model	Baseline Malware Accuracy	`MetaMe` Accuracy	Tahr Accuracy
InceptionV3	98.92%	88.16%	76.5%
VGG16	99.08%	98.33%	89.5%
ResNet50V2	96.4%	84.5%	79.33%

Table 2: CNN Performance after Adversarial Training

CNN Model	`MetaMe` Accuracy	Tahr Accuracy
InceptionV3	89.67%	87.17%
VGG16	98.83%	91.67%
ResNet50V2	89.33%	81.17%

would not reflect the primary advantage of our approach, which is enabling transformation families that procedural engines cannot easily support.

Attack Against ML Detection Algorithms. We evaluate evasion against two fundamentally different detection paradigms: *image-similarity* CNNs, which interpret binaries as visual textures and are therefore sensitive to structural perturbations in code layout, and *feature-guided* tree ensembles, which operate on extracted statistical features such as byte histograms and opcode distributions. Evaluating against both ensures that our transformations are effective across detection approaches that target different aspects of the binary. We use the balanced 8,000-sample dataset with a 70/15/15 split; all reported accuracy values are computed on the test set and implicitly account for false-positive rates.

CNN-based image-similarity models and feature-guided tree ensembles represent the two dominant paradigms in static ML-based malware detection [38, 30]. Image-based approaches capture the visual texture of binaries and are sensitive to structural layout changes, while tree ensembles operate on statistical feature vectors (byte histograms, opcode distributions, entropy) and are sensitive to distributional shifts. Together, they cover the two principal axes along which instruction-level metamorphism can perturb a binary's representation.

Detectors based on control flow graphs (CFGs) or dynamic behavioural analysis target semantic properties of the program. Since our grammar operates exclusively at the instruction level and preserves the CFG by construction (Section 3), CFG-based detectors are structurally orthogonal to the transformations we evaluate. Similarly, dynamic-analysis detectors observe runtime behaviour, which is preserved by semantic equivalence. Evaluating against these detector families, while interesting for completeness, would not test the specific evasion dimension that instruction-level metamorphism targets. We leave such evaluation to future work.

Table 3: Tree-based Models Performance Against Metamorphic Attacks

Model	Baseline Malware Accuracy	`MetaMe` Accuracy	Tahr Accuracy
RandomForest	99.83%	99.67%	93.5%
XGBoost	99.83%	100%	94.67%

CNN-detectors. CNN-based malware detection relies on image similarity, so we converted our binaries to grayscale images, a standard preprocessing step [11]. Subsequently, we trained and benchmarked three CNN models: `ResNet50v2` [19], `InceptionV3` [36], and `VGG16` [35]. The results, shown in Table 1, illustrate that all models achieve very high baseline detection rates on non-obfuscated samples.

To measure adversarial robustness, we adopted the following protocol. We randomly selected 600 malware samples from the *training* split, applied one metamorphic transformation step to each sample using both `MetaMe` and our Tahr-based engine, and added the resulting variants to the training set for adversarial training. Evasion was then evaluated on independently generated variants of samples drawn exclusively from the *test* split; no test-set sample or derivative was used during any training phase. The results in Table 1 show that while `MetaMe` reduces the accuracy of all models, particularly for `InceptionV3` and `ResNet50v2`, our grammar-guided TE achieves higher evasion rates in our experimental setting. We note that part of the observed accuracy reduction is attributable to the well-documented fragility of image-based classifiers against structural perturbations [11]; however, our grammar-driven transformations consistently outperform `MetaMe` against the same classifiers, confirming that a richer transformation space (enabled by variable-length rewrites and recursive expansions) yields greater evasion beyond what `MetaMe`'s same-size constraint permits.

To study the defensive usefulness of these samples, we included the `MetaMe`-obfuscated samples in the training set for adversarial training. The results in Table 2 show that, although the accuracy improved against `MetaMe` variants, it does not recover to levels comparable to the baseline when tested against our grammar-based variants. This result demonstrates that a CNN model trained on malware samples obfuscated by `MetaMe` remains vulnerable to more effective metamorphic transformations that our grammar-based approach supports. Notably, as shown in Table 2, the `ResNet50V2` model shows signs of overfitting during fine-tuning, severely hampering its detection capabilities.

Tree-based detectors. Feature-guided tree ensembles classify binaries by extracting statistical attributes (byte histograms, entropy, opcode distributions) rather than treating the file as an image, providing a fundamentally different threat surface.

We assessed `RandomForest` [20] and `XGBoost` [12] using the feature families most commonly adopted in the static malware detection literature [38]: byte histograms (256-bin frequency vectors over the raw binary content), opcode distributions (frequency counts of disassembled mnemonic opcodes), and struc-

Table 4: Tree-based Models Performance After Adversarial Training

Model	`MetaMe` Accuracy	Tahr Accuracy
RandomForest	100%	94%
XGBoost	100%	94.33%

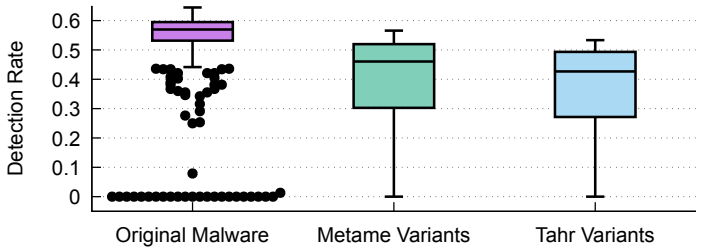


Fig. 5: Detection Rate of Malware by Virus Total

tural metadata (file size, section entropy, number of sections, and symbol-table statistics). These features capture complementary aspects of the binary: byte histograms reflect the global content distribution, opcode frequencies summarise the instruction-level composition, and structural metadata exposes format-level anomalies. We applied information-gain ranking to each model independently and retained the top-50 most discriminative features, reducing dimensionality while preserving classification power. Table 3 reports accuracy.

`MetaMe` is unable to generate adversarial examples that bypass tree-based detection (Table 3), while grammar-guided transformations reduce accuracy to 93–94%. This result shows that variable-length rewrites and recursive expansions are necessary to perturb the feature space targeted by tree-based models: `MetaMe`'s same-size substitution constraint leaves the statistical feature profile of the binary largely unchanged, so tree-based models that rely on byte histograms and opcode distributions are unaffected. Our grammar, by contrast, introduces variable-length rewrites and recursive expansions that alter the opcode distribution and byte-level structure, causing a measurable shift in the feature space. Training with `MetaMe` variants produces no relevant accuracy change, confirming that same-length substitutions provide no generalisation signal. Post-adversarial-training results are reported in Table 4.

Attack Against Signature-based Detection Systems. To evaluate the resilience of our approach against commercial signature-based defence systems, we have uploaded the 600 malware used in the ML experiments to VirusTotal, so as to rely on multiple commercial antivirus systems and sandboxes. For each malware, we have generated three variants: an unmodified baseline, a variant generated by `MetaMe`, and a variant obfuscated using our Tahr-based TE.

The distribution of detection rates is shown in Figure 5. We observe some outliers in the baseline distribution, resulting in a detection rate of 0. These anomalies likely represent artefacts in the VirusShare dataset, or samples that have become dormant, irrelevant or unclassifiable by modern antiviruses.

The comparative analysis shows that Tahr variants achieve a clear reduction in detection rate compared to the baseline and `MetaMe`. The non-overlapping interquartile ranges of the Tahr and baseline distributions in Figure 5 indicate that this difference is consistent across the sample population and not driven by outliers. The lower bounds of the distribution converge, demonstrating successful total evasion for a non-negligible fraction of the samples. At the same time, Tahr variants show a lower median and a compressed upper quartile, indicating that even when our approach is unable to achieve total evasion, it successfully deceives a larger number of antivirus engines than the state-of-the-art `MetaMe` alternative.

5 Mitigations

Having established that grammar-based transformations evade both baseline and `MetaMe`-hardened classifiers, we ask whether retraining on grammar-generated variants can immunise models against this threat. We selected the most robust models from the previous phase (`VGG16`, `RandomForest`, and `XGBoost`), excluding `ResNet50V2` due to its overfitting instability, and performed adversarial training using metamorphic variants from the training set.

The results of this analysis are reported in Table 5. They show a significant improvement in detection accuracy, with tree-based models achieving 100% accuracy against both our engine's variants and `MetaMe` samples. Reaching 100% accuracy should not be misinterpreted as the metamorphic transformation leaving some form of explicit fingerprint or recognisable structural artifact: a grammar-based transformation cannot produce a consistent fingerprint by construction, provided the grammar contains multiple overlapping productions for the same nonterminal symbol: each generated variant then results from a different stochastic path through the rule space. In the degenerate case where every nonterminal has a single production, the transformation would be deterministic and therefore fingerprintable. The grammar used in our experiments is specifically designed with multiple alternative productions for each matchable instruction category, ensuring that every transformation pass samples from a combinatorially large space of possible rewrites and thus prevents the emergence of a stable signature. Rather, the adversarial training process forces tree-based algorithms to perform a dynamic *feature realignment*: exposing the classifier to the grammar's transformation space forces the feature selection mechanism to converge on the invariant characteristics of the malicious payload that lie outside the grammar's scope. This result suggests that the models did not merely memorise specific implementation artifacts, but learnt to identify the features that the grammar cannot systematically alter.

The data support the conclusion that feature-guided tree-based models are more resilient to metamorphic attacks compared to image-similarity approaches.

Table 5: Metamorphic attack detection performance via meta-grammar training

Model	`MetaMe` **Accuracy**	**Tahr Accuracy**
RandomForest	100%	100%
XGBoost	100%	100%
VGG16	97.67%	97%
InceptionV3	88.17%	86.83%

This result can be related to their ability to extrapolate the perturbations introduced by the grammar-based rules, and exploit them during the classification.

A possible alternative interpretation of the tree-based models' 100% post-training accuracy is that they learn to detect artifacts introduced by the grammar itself rather than malicious semantics. We consider this unlikely for two reasons. First, the grammar is non-deterministic: no two generated variants share a fixed structural fingerprint, and stochastic rule selection produces variants that differ in length, opcode distribution, and instruction composition. Second, adversarial training used variants from the *training* set while evaluation used independently generated variants of the *test* set, so the reported accuracy reflects generalisation across independent samples, not memorisation of a fixed artifact. Nevertheless, verifying generalisation to metamorphic techniques that lie outside the grammar's transformation space remains an important direction for future work. We identify two concrete experiments. First, holding out entire grammar rule families (e.g., all expansion rules or all register-remapping productions) during adversarial training and testing on variants produced by the held-out rules would reveal whether the models generalise across rule families rather than memorise specific transformation patterns. Second, evaluating the adversarially trained models on variants produced by fundamentally different engines (e.g., `MetaMe` variants, hand-crafted metamorphic samples, or RL-generated perturbations [31, 2]) would test cross-engine generalisation and provide stronger evidence that the learned features capture genuine malicious semantics rather than grammar-specific artifacts.

It is important to note that we have assumed a *blind* attacker in our evaluation: an analysis of a *feature-guided* adversary, who specifically targets the learned features of these hardened models, remains outside the scope of this work.

Threats to Validity. The main threat to internal validity is the reliance on a subset of the x86 instruction set: our grammar does not cover the complete ISA, and instructions that match no production rules are passed through unchanged. A binary composed predominantly of unmatched instructions would not be meaningfully transformed. However, the instructions in our grammar account for the majority of instructions found in compiled C binaries, and the experimental evaluation is based on real-world malware samples rather than synthetic ones.

A threat to external validity is the choice of VirusShare 2019-2020 as the malware corpus. While it is a widely used benchmark, the detection rates we observe on VirusTotal may differ from those on more recent corpora or on samples from different malware families. The dataset is also limited to Linux ELF binaries

compiled for Intel x86-64, so the results should not be generalised to other platforms or binary formats without further validation. Moreover, the benign corpus (GNU binutils) differs in compilation toolchain and software domain from the VirusShare malware samples, which may introduce distributional artifacts such as different compiler idioms, optimisation levels, or library dependencies. This is a standard limitation of binary classification experiments where ground-truth malware and benign corpora originate from different sources, and it should be considered when interpreting the absolute accuracy figures.

Although we use a single dataset, overfitting is mitigated by three factors: (1) we fine-tune pre-trained CNN models via transfer learning rather than training from scratch, which regularises the feature extractors; (2) we use a 70/15/15 split with a held-out test set that is never used during training or hyperparameter selection; and (3) all results reported in the paper are computed on the test set. Additionally, no temporal analysis of sample first-appearance dates was performed, and the 1:1 ratio of benign to malicious samples in the training set may not reflect real-world class priors. Both factors should be considered when extrapolating the results to deployment scenarios.

The formal semantic verification using `Alive2` applies only to the transformation rules and not to the final mutated binaries as a whole: the tool's limitations (inability to handle stack-altering transforms and conditional jumps) prevent full-binary verification. Nevertheless, the successfully verified rules cover the most commonly applied transformations in our grammar, and the `revert` mechanism ensures that attribute constraints are satisfied at runtime for all rules, including those excluded from formal verification.

6 Related Work

While metamorphism is not novel, the methodology for implementing it has remained significantly stagnant. The field is heavily influenced by [37], which introduced the first modular metamorphic engine based on discrete, hard-coded substitutions and established the standard expand-and-reduce workflow [39] that we retain. Advanced engines such as MetaPHOR and W32.Zmist later extended this procedural model with recursive expansion pipelines and multi-stage mutation strategies. These engines demonstrate that the individual building blocks of metamorphism (register remapping, instruction substitution, expansion and reduction) have been understood for decades. Our contribution is not to introduce new transformation families, but to reformulate them declaratively via attribute grammars, obtaining composability, stochastic diversity, and formally checkable semantic constraints that procedural implementations cannot easily guarantee. As noted in [8], a static substitution database may not suffice for acceptable obfuscation; grammar-driven non-determinism addresses this directly.

Table 6 summarises the key dimensions along which our grammar-based engine differs from related approaches. Our approach is the only one combining a formal specification via attribute grammars, provable semantic constraints enforced through attribute evaluation and automatic revert, stochastic diversity from non-

Table 6: Qualitative Comparison of Metamorphic and Adversarial Binary Transformation Approaches

Approach/Tool	Target	Transformation Types	Formal Spec.	Semantic Guarantees	Stochastic	Open Source
Our approach	x86-64 ELF	Subst., remap, expansion, reduction	Yes	Attribute constraints + revert	Yes	Yes
`MetaMe` [28]	x86-64 ELF	Same-size substitution	No	Manual	No	Yes
MetaPHOR	x86 (Win32)	Subst., expansion, reduction	No	None	Partial	No (malware)
Tigress [6]	Source-level (C)	CFG flattening, virtualisation, opaque pred.	No	Compiler-level	Yes	Yes
ADVeRL-ELF [31]	ARM ELF	Semantic NOP insertion	No	RL reward	No	Partial
LLM-based [24]	Various	Syntactic rewriting	No	None (probabilistic)	Yes	N/A
OLLVM	LLVM IR	CFG flattening, substitution	No	Compiler-level	No	Yes

deterministic rule selection, and open-source availability, all targeting x86-64 ELF binaries at the binary level. Source-level tools such as Tigress [6] and OLLVM operate at a fundamentally different point in the software distribution pipeline, requiring access to source code or LLVM intermediate representations that are unavailable for deployed binaries and captured malware samples. ADVeRL-ELF [31] employs a single transformation technique (semantic NOP insertion) on a different ISA (ARM), limiting both the breadth of the transformation space and direct comparability with x86-64 engines. LLM-based approaches [24] offer syntactic diversity through the stochastic nature of language model generation, but cannot guarantee semantic preservation, as correctness depends on the probabilistic output of the model rather than on enforceable formal constraints.

A seminal contribution to the formalisation of code obfuscation is provided in [17], where the author maps mutation engines to the Chomsky hierarchy, showing that metamorphic transformations correspond to context-sensitive (type 1) or unrestricted (type 0) grammars, and that the detection problem for type 0 languages is undecidable. We follow this theoretical path, presenting a concrete formalisation for modern x86-64 architectures using AGs, which extend context-free grammars with semantic constraints and achieve the expressive power envisioned in [17].

A parallel line of research has explored two-level Van Wijngaarden grammars [18, 3], which offer Turing-complete expressiveness but introduce a complexity that makes semantic correctness verification intractable. AGs allow enforcing strict semantic preservation with a simplified encoding of transformations.

Recently, in [24], Large Language Models (LLMs) have been proposed to drive metamorphic transformations, confirming their potential for syntactic diversity.

However, the probabilistic nature of LLMs makes functional correctness difficult to guarantee, a limitation that AGs avoid by construction.

A distinct family of related tools consists of *source-level* and *compiler-level* obfuscators such as Tigress [6] and OLLVM. These tools apply transformations (control flow flattening, opaque predicates, virtualisation) at the source or LLVM IR level and require access to the program's source code or its intermediate representation. Our engine operates directly on compiled ELF binaries using disassembly and reassembly, with no source code required, targeting a different point in the software distribution pipeline where source code is unavailable. Commercial virtualisation-based obfuscators such as VMProtect and Themida apply code virtualisation to Windows PE binaries, a technique orthogonal to instruction-level metamorphism and targeting a different platform, so direct experimental comparison with these tools is not the objective of this work.

In adversarial malware generation, several approaches are closely related to our evaluation context. [41] introduced an evolutionary framework for evasion of PDF malware classifiers, demonstrating that automated mutation can reduce classifier accuracy to near zero without breaking malicious functionality. [2] applied reinforcement learning to mutate Windows PE headers and sections, showing that a black-box RL agent can learn functionality-preserving evasion strategies. More recently, [31] proposed ADVeRL-ELF, an RL-based framework targeting ELF binaries on ARM architectures via semantic NOP insertion, achieving a 59.5% attack success rate against a ResNet18-based detector. ADVeRL-ELF employs a single transformation technique, the injection of semantically neutral instruction sequences (semantic NOPs), and does not perform instruction substitution, register remapping, or recursive expansion and reduction of instruction sequences. These operations constitute the core of classical metamorphism and are the focus of our grammar. Comparing against ADVeRL-ELF would therefore conflate two distinct transformation paradigms, semantic NOP insertion and metamorphic rewriting, without providing insight into the effectiveness of the grammar formulation for metamorphic engines. An x86-64 adaptation of ADVeRL-ELF would require re-engineering both the RL reward function and the action space for a different ISA and a fundamentally different transformation model, which constitutes an independent research effort beyond the scope of this paper. Investigating RL-guided rule selection within our grammar framework, where the RL agent selects among grammar productions rather than NOP insertion points, is an interesting direction for future work. [16] addresses functionality-preserving attacks on Windows PE files through black-box optimisation of header content, a complementary attack surface to instruction-level rewriting. The problem-space framework of [29] provides a formal characterisation of the gap between feature-space attacks and realisable problem-space perturbations, a distinction directly relevant to our approach: our transformations are problem-space attacks by construction.

Finally, the work in [15] demonstrates that metamorphic techniques can be used in defence contexts. The authors apply mutation rules to YARA byte signatures to anticipate obfuscated variants of known malware families. Our work

confirms the applicability of metamorphic techniques to defence scenarios, but, while in [15] the authors mutate the detector (i.e., the static rules), we mutate the data for adversarial training. Our results show that *defensive metamorphism* can improve the accuracy of predictive models against complex mutations. This is a complementary and practically important observation: the grammar-based engine, designed as an offensive tool to generate novel metamorphic variants, can be directly repurposed to generate the adversarial training data needed to harden detectors against exactly the class of transformations it supports.

7 Conclusions and Future Work

We have presented a declarative reformulation of metamorphic binary transformation using translational attribute grammars. By encoding well-known transformation families (instruction substitution, register remapping, expansion and reduction) as grammar productions with associated semantic attributes, we obtain a transformation engine that is composable, stochastically diverse, and semantically constrained by construction. The grammar's three-level hierarchy (syntactic, semantic, translational) separates transformation intent from instruction-level encoding, providing a modular structure that simplifies adaptation to different target ISAs.

Our evaluation on 4,000 Linux ELF malware samples shows that grammar-guided variants achieve higher evasion rates than `MetaMe` across all evaluated classifiers: CNN-based image-similarity models, feature-guided tree ensembles (`RandomForest`, `XGBoost`), and commercial antivirus engines via VirusTotal. Grammar-generated variants are also effective for adversarial training: models retrained on grammar-mutated samples achieve near-perfect detection accuracy, whereas the same retraining with `MetaMe` variants yields no improvement. The declarative structure of the grammar enables a rule-by-rule validation methodology, applied here via bounded translation validation with `Alive2` and `ReOPT` on the core transformation rules.

Future work will focus on evolving the attacker paradigm from a blind one to a context-aware one. We also plan to study evolutionary approaches to optimise the grammar's production capabilities against specific target classifiers, and to investigate whether grammar-guided adversarial training generalises to metamorphic techniques outside the grammar's transformation space, such as RL-generated or hand-crafted variants.

Bibliography

[1] Alam, S., Horspool, R.N., Traore, I., Sogukpinar, I.: A framework for metamorphic malware analysis and real-time detection. Comput. Secur. **48**(C), 212–233 (Feb 2015)

[2] Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P.: Learning to evade static PE machine learning malware models via reinforcement learning. arXiv preprint arXiv:1801.08917 (2018). https://doi.org/10.48550/arXiv.1801.08917

[3] Augusto, L.M.: The van wijngaarden grammars: A syntax primer with decidable restrictions. Journal of Knowledge Structures and Systems **4**(2), 1–39 (2023)

[4] Babaagba, K.O., Wylie, J.: An evolutionary based generative adversarial network inspired approach to defeating metamorphic malware. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation. ACM, New York, NY, USA (Jul 2023)

[5] Babaagba, K.O., Tan, Z.: Mitigating metamorphic malware through adversarial learning techniques. Preprints (Sep 2025)

[6] Banescu, S., Collberg, C.S., Pretschner, A.: Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In: Proceedings of the 26th USENIX Security Symposium. pp. 661–678 (2017)

[7] Borello, J.M., Mé, L.: Code obfuscation techniques for metamorphic viruses. Journal in Computer Virology **4**(3), 211–220 (Aug 2008)

[8] Brezinski, K., Ferens, K.: Metamorphic malware and obfuscation: A survey of techniques, variants, and generation kits. Secur. Commun. Netw. **2023**(1), 1–41 (Sep 2023)

[9] Caliandro, P., Ciccaglione, M., Pepe, A., Bianchi, G., Pellegrini, A.: Vmorph: A virtualization/metamorphic framework for binary obfuscation and intellectual property protection (2025)

[10] Capstone: Capstone: The ultimate disassembler framework (2020)

[11] Catalano, C., Chezzi, A., Angelelli, M., Tommasi, F.: Deceiving AI-based malware detection through polymorphic attacks. Comput. Ind. **143**(103751), 103751 (Dec 2022)

[12] Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 785–794. ACM, New York, NY, USA (Aug 2016). https://doi.org/10.1145/2939672.2939785

[13] Ciccaglione, M., Caliandro, P., Pellegrini, A.: Tahr: The generative attribute grammar framework. arXiv [cs.FL] (Dec 2025)

[14] Ciccaglione, M., Caliandro, P., Pellegrini, A.: Tahr: The generative attribute grammar framework (Dec 2025)

[15] Coscia, A., Dentamaro, V., Galantucci, S., Maci, A., Pirlo, G.: YAMME: A YAra-byte-signatures metamorphic mutation engine. IEEE Trans. Inf. Forensics Secur. **18**, 4530–4545 (2023)

[16] Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A.: Functionality-preserving black-box optimization of adversarial windows malware. IEEE Transactions on Information Forensics and Security **16**, 3469–3478 (2021). https://doi.org/10.1109/TIFS.2021.3082330

[17] Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. canInternational Journal of Electrical and Computer Engineering **2**(1) (Mar 2007)

[18] Geoffroy, G.: Van wijngaarden grammars and metamorphism. In: 2011 Sixth International Conference on Availability, Reliability and Security. pp. 466–472. IEEE (Aug 2011)

[19] He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: Computer Vision – ECCV 2016, pp. 630–645. Lecture Notes in Computer Science, Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-46493-0_38

[20] Ho, T.K.: The random subspace method for constructing decision forests. IEEE Transactions on Pattern Analysis and Machine Intelligence **20**(8), 832–844 (1998). https://doi.org/10.1109/34.709601

[21] Keystone: Keystone (2016)

[22] Koppanati, R.K., Peddoju, S.K.: MSG: Missing-sequence generator for metamorphic malware detection. J. Inf. Secur. Appl. **89**(103962), 103962 (Mar 2025)

[23] Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: bounded translation validation for LLVM. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. vol. 1. ACM, New York, NY, USA (Jun 2021)

[24] Madani, P.: Metamorphic malware evolution: The potential and peril of large language models. In: 2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA). pp. 74–81. IEEE (Nov 2023)

[25] Maestre Vidal, J., Sotelo Monge, M.A.: Obfuscation of malicious behaviors for thwarting masquerade detection systems based on locality features. Sensors (Basel) **20**(7), 2084 (Apr 2020)

[26] Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). IEEE (Dec 2007). https://doi.org/10.1109/acsac.2007.4413008

[27] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[28] Pellitteri, A.: MetaMe: A metamorphic code engine. https://github.com/a0rtega/metame (2017), accessed: 2024-5-22

[29] Pierazzi, F., Pendlebury, F., Cortellazzi, J., Cavallaro, L.: Intriguing properties of adversarial ML attacks in the problem space. In: Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P). pp. 1308–1325 (2020). https://doi.org/10.1109/SP40000.2020.00073

[30] Polamarasetti, A.: Research developments, trends and challenges on the rise of machine learning for detection and classification of malware. In: 2024 International Conference on Intelligent Computing and Emerging Communication Technologies (ICEC). pp. 1–5. IEEE (Nov 2024). https://doi.org/10.1109/icec59683.2024.10837413
[31] Ravi, A., Chaturvedi, V., Shafique, M.: ADVeRL-ELF: ADVersarial ELF malware generation using reinforcement learning. In: Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC) (2025). https://doi.org/10.1109/DAC63849.2025.11132466
[32] Roberts, J.: VirusShare. https://virusshare.com/ (2011), accessed: 2024-5-22
[33] Scott, R.G., Boston, B., Hendrix, J., Ravitch, T., Wagner, A.S., Davis, B., Diatchki, I., Dodds, M., Matichuk, D., Quick, K., Robert, V., Selfridge, B., Wagner, D., Winwood, S.M.: M : A machine code toolbox for the busy binary analyst (Feb 2025)
[34] Scott, R.G., Winwood, S.: M : A machine code toolbox for the busy binary analyst (Jul 2024)
[35] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (Sep 2014)
[36] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE (Jun 2016). https://doi.org/10.1109/cvpr.2016.308
[37] The Mental Driller: Metamorphism in practice or how I made MetaPHOR and what I've learnt. Tech. rep., 29A (Feb 2002)
[38] Ucci, D., Aniello, L., Baldoni, R.: Survey of machine learning techniques for malware analysis. Computers & Security **81**, 123–147 (Mar 2019). https://doi.org/10.1016/j.cose.2018.11.001
[39] Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: The design space of metamorphic malware (2007)
[40] Wu, Z., Gianvecchio, S., Xie, M., Wang, H.: Mimimorphism: a new approach to binary code obfuscation. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 536–546. CCS '10, Association for Computing Machinery, New York, NY, USA (Oct 2010)
[41] Xu, W., Qi, Y., Evans, D.: Automatically evading classifiers: A case study on PDF malware classifiers. In: Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS) (2016). https://doi.org/10.14722/ndss.2016.23115
[42] Zou, X., Gong, X., Zhang, J., Li, S., Yew, P.C.: XuanJia: A comprehensive virtualization-based code obfuscator for binary protection. arXiv [cs.CR] (Jan 2026)