ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov1219

DISSECT: AI-Driven Malware Deobfuscation and Semantic Reconstruction for Python Code

Dr. Vijayashekhar S. S.[1]; Jeffson Chandy Joseph[2]; Joyson N. W.[3]; Kiran Kumar K.[4]; Nanda Kishor U. K.[5]

[1]Department of AI & ML Acharya Institute of Technology Bangalore, India
[2]Department of AI & ML Acharya Institute of Technology Bangalore, India
[3]Department of AI & ML Acharya Institute of Technology Bangalore, India
[4]Department of AI & ML Acharya Institute of Technology Bangalore, India
[5]Department of AI & ML Acharya Institute of Technology Bangalore, India

Publication Date: 2026/11/28

Abstract: Python code is often obfuscated to make it harder to analyze. Identifier renaming, literal encoding, insertion of more code statements, and flattening of control flow are some methods that make the process of analyzing the code difficult and obscure its intended purpose. In this paper, we introduce DISSECT, a modular framework for static inspection and assisted reconstruction of obfuscated Python source code. The framework consists of lexing, abstract syntax tree inspection, control flow graph creation, obfuscation pattern detection, rule-based transformation, optionally sequence-to-sequence reconstruction, syntax fixing, confidence estimation, and analysis persistence capabilities. It also includes support for manual submissions as well as automatic discovery of files through a background orchestration pipeline. The provided benchmark consists of 100 benign base programs and four artificial variants per program, which results in 400 obfuscated samples. A test partition of 60 variants is also provided. The obtained results show 100% syntax validity, token F1 score of 0.4286, CodeBLEU of 0.3807, Abstract Syntax Tree similarity of 0.6066, and Control Flow Graph similarity of 0.834. However, all five ablation configurations provide the same metrics, and the evaluation manifest shows the rule-based fallback was used as the engine without initializing pretrained model weights.These findings thus reflect the current baseline run and do not prove that neural reconstruction or structural conditioning enhance performance. The present paper covers the architecture, implementation, evaluation, limitations, and practical safety concerns without using the results obtained from the synthetic benchmark as proof of effectiveness on realworld malware.

Keywords: *Abstract Syntax Tree, Cybersecurity, Code Reconstruction, Confidence Estimation, Control-Flow Graph, Python Deobfuscation, Static Analysis.*

How to Cite: Dr. Vijayashekhar S. S.; Jeffson Chandy Joseph; Joyson N. W.; Kiran Kumar K.; Nanda Kishor U. K. (2026) DISSECT: AI-Driven Malware Deobfuscation and Semantic Reconstruction for Python Code. *International Journal of Innovative Science and Research Technology*, 11(11), 2109-2115. https://doi.org/10.38124/ijisrt/26nov1219

I. INTRODUCTION

Obfuscation refers to the purposeful modification of a program to render it more difficult to comprehend while maintaining some or all of its original functionality. Obfuscation serves many different purposes from protection of intellectual property to malicious use in an attempt to hide suspicious behavior. Python programming language is especially interesting to defensive examination due to its readability, dynamic characteristics of its execution environment and the ability to distribute code along with any other content in an encoded format. Adversary TTP documented in the MITRE ATT&CK project may also be useful to defensive examination of obfuscation, even though this paper focuses on reconstruction of source code [1].

Obfuscation techniques are renaming variables/functions, string literal encoding, adding irrelevant code, using indirect expressions where direct ones could have been used, and turning structured control flow into a dispatcher loop. One particular transformation is simple to deobfuscate, but many stacked together will make the job hard. Moreover, a formatter application may make the code readable without retrieving the original meaning. Thus, an effective analysis approach must state clearly what it is able to retrieve, where there is doubt, and differentiate between valid syntax and semantics.

In this paper, DISSECT is presented which is a modularitybased python-centric approach integrating static program analysis and deobfuscation assisted approaches. It has the following implementation elements such as source

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov1219

code parsing, lexical and Abstract Syntax Tree (AST) features, Control Flow Graph (CFG) creation, obfuscation determination, strategy identification, rule-based path for recovery, an optional transformer-based model manager, syntax repair, semantic and indicator extraction, confidence calculation, reporting, and job tracking via database. There is also an approach to discover files.

The main design philosophy is that we examine the submitted source code as data and not run it as part of our static analysis process. This minimizes the risk of exposure to untrusted program behaviors; however, it does not make the whole software stack safe since we still need normal security measures for parsers, dependencies, archives, and file processing routines. Behavioral comparisons used in the evaluation section are distinct and only use benchmark programs.

➢ *Objectives*

The project has five practical objectives:

- Identify common lexical, syntactic, and control-flow signs of obfuscation.
- Recover transformations that can be reversed using constrained static rules.
- Use AST and CFG summaries to support interpretation and, when available, model-assisted reconstruction.
- Assess reconstructed output using several complementary checks rather than one score.
- Provide reports and confidence information that help an analyst decide what to review next.

➢ *Contributions and Scope*

The key contribution is an implementation-oriented description of the workflow of combined analysis, including data preparation and evaluation artifacts. The authors do not claim that their current experiments provide evidence of production readiness, universal semantic equivalence, or effectiveness against advanced real-world malware. The benchmark is artificial and benign, and the provided run included a rule-based fallback. These constraints carry through to the results and conclusion.

II. BACKGROUND AND RELATED WORK

➢ *Static Program Representations*

Static analysis is done on the source code or representation of the source code from the submitted code without running it. The Python AST represents the expressions, assignments, branching, looping, and function definitions of the source code in structured forms of the nodes according to the official documentation of Python [2]. We can use it for analyzing relationships which are difficult to determine via text matching. The CFG augments the AST by describing possible control flow transfers between basic blocks.

Neither is a comprehensive representation of the program behavior under all circumstances. In Python, it is possible to have reflection, dynamic imports, runtime code generation, and environment interaction. A static analysis tool can thus find suspicious constructs, make a good approximation of the program structure, and yet not be able to predict all effects at run time. DISSECT considers AST and CFG results as hints for the analysis but not proof of full reconstruction.

➢ *Deobfuscation and Code Models*

Classical methods to remove obfuscations rely on patterns of transformations, logical reasoning, and/or simplifications of programs. For example, Syntia reconstructs expressions by synthesis [3]. Rule-based technique works well if a given obfuscation has a known structure. However, if there is a transformation pattern that was not seen before or the obfuscation relies on its execution time, rule-based approach fails.

Code models that are pre-trained represent yet another direction. CodeT5 is an encoder-decoder model pre-trained especially for code-based tasks including generation and transformation tasks [4]. However, models like these can create fluent reconstructions; however, fluency does not always mean accuracy. The generated code can be parsed without errors while altering a condition, skipping an operation, or creating an API call. This is why DISSECT pairs up model output with structural validation and reportage. In the present experiment, however, no pre-trained weights were used, which leaves the question of the model contribution open.

➢ *Evaluation and Confidence*

CodeBLEU augments traditional text overlap measures by adding code-specific metrics such as syntax and data flow information [5]. Token F1, AST similarity, CFG similarity, syntax validity, exact match, and behavior comparison all measure different dimensions of the reconstruction effort. There is no single metric that can be viewed as being adequate in itself to measure correctness. Calibration can be applied in conveying uncertainty, but it will only be feasible in the framework of the training and assessment of the calibration.

A. System Requirements and Design Principles

There are a number of requirements for DISSECT. First, it should allow for accepting code either pasted by the user or provided as a file. Second, the system should allow for keeping the analysis stages as separate modules in order to allow for inspection of detection, reconstruction, and reporting separately. Third, the output should contain both a reconstructed candidate and evidence. Fourth, a failure to load the model should not silently block all analysis; there is a deterministic rule-based fallback mechanism provided.

The boundary of static analysis is significant. The submitted source code is parsed and converted as text and syntax constructs; it is not intentionally executed by the static analyzer. If any behavioral tests are conducted during the evaluation, they have to be conducted through a different harness, which has to be limited, and benign programs verified by the authors have to be used.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov1219

III. DISSECT ARCHITECTURE

Fig. 1 is the summary of the primary processing flow. In terms of implementation, the architecture is modular, where the API takes in the request, the pipeline manages the processing stage, the modules handle the representation and indicator extraction, and the reporting layer handles output. Discovery process is automatic, and follows a parallel entry route to the same orchestration service.

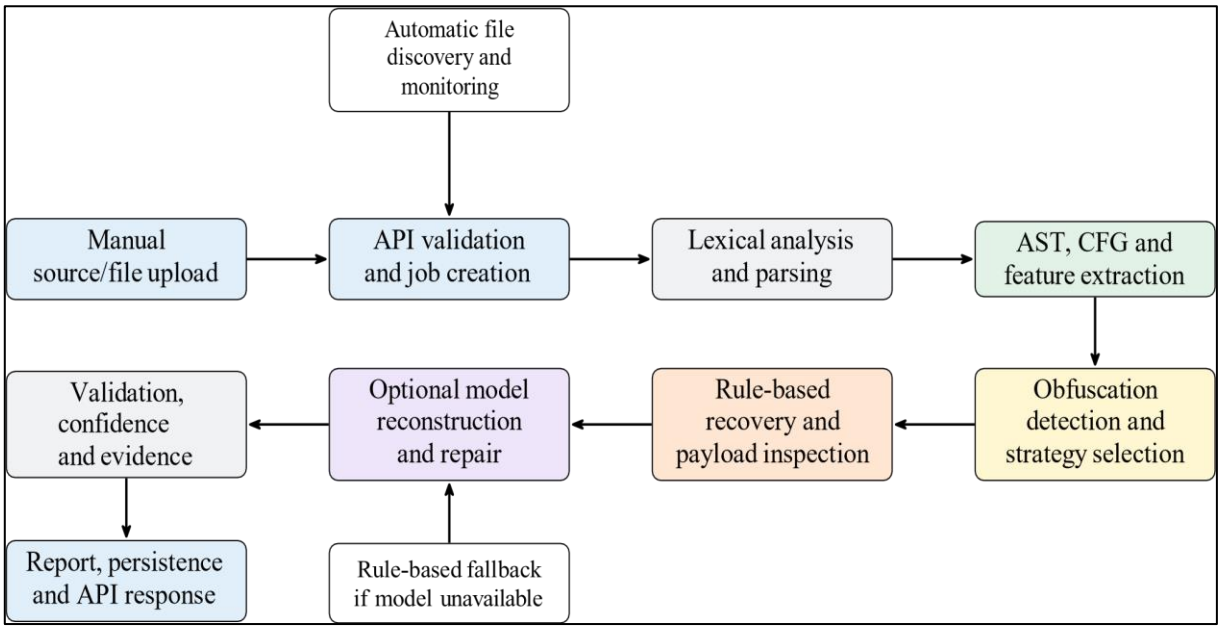


Fig 1 High-Level DISSECT Workflow Based on the Supplied Source Modules. The Model Stage is Optional and Can Fall Back to Deterministic Rules.

➢ *Input and Orchestration*

These API endpoints provide support for analyzing requests and file scanning processes. The orchestration module is responsible for handling background tasks, limited concurrency of workers, deduplication and persistence. File hashes may be used to identify an already analyzed file, which makes no need for re-analyzing the same content again. Job state change events allow monitoring of job status both in queue and finished jobs.

➢ *Lexical and AST Analysis*

The lexical analyzer looks at the properties and characteristics at the level of tokens that could be linked to obfuscation. The AST analyzer operates on the parsed structure of Python to analyze statements, expressions, names, calls, and other structures. The feature extractor collects properties that will be used for classification and confidence later on. Parser errors also hold significance as a source of evidence since a file that cannot be parsed as regular Python source code could be incomplete, formatted differently, or have a wrapper.

➢ *CFG Generation*

CFG generation is based on basic blocks and their controlflow dependencies. Such an approach works well when a transformation leads to a change from a simple branch to a dispatcher loop or adds new branches. CFG similarity is one of the criteria for the comparison in the evaluation phase. CFG generation from the source code is an approximation in cases where Python's dynamic capabilities are involved.

➢ *Detection and Strategy Selection*

The detector identifies several types of obfuscations and gives results that will help in the next phase. The strategy selector selects from the available recovery techniques. The separation of roles is beneficial since the tasks of decoding literal, simplifying expression, and recovering flattened control flow graph are distinct operations. The result obtained by the detector must be interpreted as a hint about the presence of a pattern and not as evidence of the malice of the program.

➢ *Reconstruction and Repair*

Deterministic transforms are performed by the rule-based baseline in cases when the inputs match the supported patterns.

Abstract evaluator and expression reducer are designed to simplify the expressions without actually running the submitted code. Modules for payload extraction and structure reconstruction deal with the wrapped or nested source representations. Model manager allows loading of a configured sequence-tosequence model and acts as a fallback option in case if there is no possibility to initialize and infer the model.

Fixing should be undertaken carefully. Just because the source parses does not mean that there will not be changes to the semantics of the source. That is why syntax repair is considered as the stage preceding additional verification. It is important to maintain the distinction between the recovered source, repair actions and their justification at the reporting stage.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov1219

➢ *Semantic Interpretation and Reporting*

The semantic modules examine behavior indicators, detect potential indicators of compromise(IOCs), match any results to relevant threat intelligence, and provide human-readable rationale and triage data. The reports should separate observable facts from analysis and interpretation. A string that looks like a URL, a file system API function call, or a dynamic execution primitive should be examined, but not because it proves anything on its own.

A. Deobfuscation Workflow

The system's processing can be explained as six stages, shown in Fig. 2.

➢ *Stage 1: Safe Input Handling*

Input layer takes source text or a file name as an input and validates the input request. File discovery could identify potential files and pass them on to the orchestrator. Restrictions on file sizes, access control, timeouts, updating dependencies, and isolated deployment are necessary safeguards. Static content alone removes the threat of execution but is not without its weaknesses.

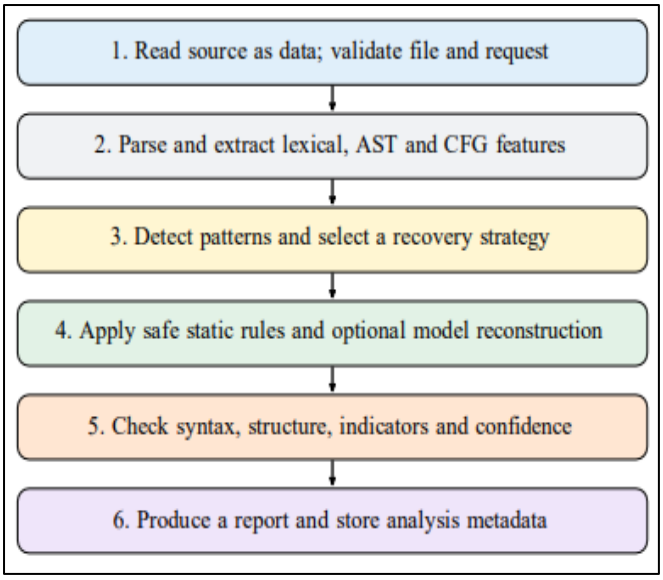


Fig 2 Six-Stage View of the DISSECT Workflow.

➢ *Stage 2: Representation building*

The input is tokenized and parsed where feasible. There are two kinds of representations for the code: AST and CFG. This allows the system to extract features like names, calls, constants, branching structures, and suspicious wrappers. The representation will help in making future decisions about the code.

➢ *Stage 3: Pattern detection*

The detection process looks for patterns like encoded literals, abnormal identifiers, dynamic execution mechanisms, dispatch loop structures, and structural anomalies. The output may contain more than one finding since actual code can contain a combination of different techniques. The detection device is a triage instrument, not a yes/no malware determination system.

➢ *Stage 4: Recovery*

Transformation through the rule-based approach comes first before any other transformation attempt. This involves, depending on the type of the input, simplifying the expression, decoding the literal under certain constraints, extraction of the embedded source, or re-writing structures. If there is a configuration of a model that exists, then the model manager attempts to reconstruct the sequence-to-sequence using the source and possibly structural summaries. If loading does not work or weights are missing, then the rule-based fallback is still an option.

➢ *Stage 5: Validation and confidence*

The output is verified for syntax correctness and compared against reference code using tokens, AST, CFG, and behavior when ground truth is available. The confidence measure represents all evidence and uncertainty associated with it. This should not be considered to mean anything like the probability of the file being malware. If validated for its context, the high value of reconstruction confidence would indicate the likelihood of reconstruction matching the reference code.

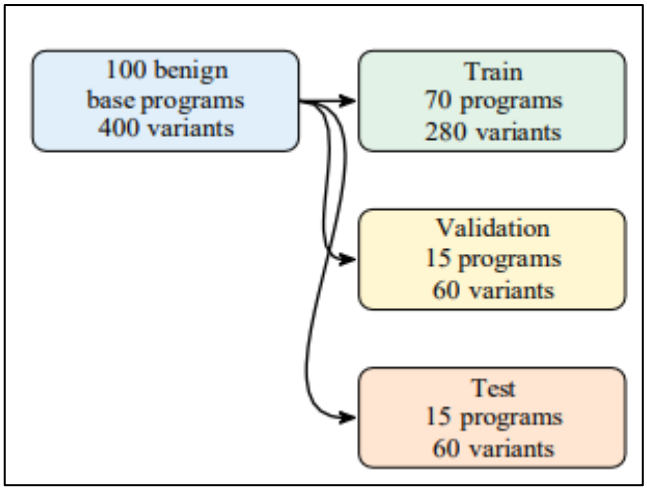


Fig 3 Base-Program Grouped Partition Reported by the Supplied Dataset Configuration.

➢ *Stage 6: Reporting*

The final report may include the metadata of the input data, detected patterns, recovered source, repair comments, extracted indicators, structural summaries, confidence information, and evaluation results. The persistence and tracking of the process help with future review. Reports should make clear the missing steps and fallback use to prevent misunderstanding a partial analysis as a reconstruction.

IV. DATASET CONSTRUCTION AND EXPERIMENTAL SETUP

➢ *Synthetic Corpus*

The provided data generation process begins with 100 benign Python code algorithms. These initial codes have four synthetic obfuscations: ast_transform, tokenize_transform, custom_flattening, and pyarmor_simulation. There are 400 obfuscated samples in total. These transformations are

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov1219

artificial and cannot capture the entire range of commercial packers or malware attacks.

These transformations include a number of general classes. The AST modifications change the code or the names in the code, as well as might add some meaningless code logic. The token transformations use the literal coding or change its notation. The custom flattening is implemented with dispatcher style control flow. And finally, PyArmor inspired test is just an implementation of a wrapper/packed-payload class.

➢ *Data Partitioning*

The partitioner groups the samples according to the identity of their base program in order for the variants which have evolved from the same base program to not end up in train, validation, and test partitions. The configuration given consists of 70 base programs for training, 15 for validation and 15 for testing. These relate to 280, 60, and 60 variants, respectively.

Table 1 Reported Aggregate Metrics on The 60-Sample Test Set

Configuration	Syntax	Tok. F1	CodeBLEU	AST	CFG
Transformer only	100%	0.4286	0.3807	0.6066	0.8340
Transformer + AST	100%	0.4286	0.3807	0.6066	0.8340
Transformer + CFG	100%	0.4286	0.3807	0.6066	0.8340
Transformer + AST + CFG	100%	0.4286	0.3807	0.6066	0.8340
Full pipeline	100%	0.4286	0.3807	0.6066	0.8340

➢ *Model and Training Status*

The model used in this configuration is Salesforce/codet5p-220m, which is an encoderdecoder code model. The training summary provided along with the project shows a smoke test run of just one epoch, which consists of two training instances and one validation instance. In both cases, the validation syntax validity score and exact match score were both zero. This is not enough to assert that a task-specific model has been trained. More importantly, the evaluation manifest clearly indicates that the pretrained weights were not initialized in this held-out evaluation, and the rule-based reconstruction engine was called.

➢ *Evaluation Metrics*

The evaluation artifacts report the following measures:

- Syntax validity: the proportion of reconstructed outputs accepted by the Python parser.
- Exact match: the proportion of outputs that exactly match their reference after the evaluator's normalization policy.
- Token F1: the balance of token-level precision and recall relative to the reference.
- CodeBLEU: a code-oriented similarity score combining multiple source-level signals [?].
- AST similarity: a comparison of structural features in the candidate and reference syntax trees.
- CFG similarity: a comparison of graph-level control-flow structure.
- Behavioral match: agreement on tested outputs and exit behavior for benign programs in the evaluation harness.

These techniques test different criteria. Parsing techniques test the criterion of parseability, while similarity scores test the criterion of semantic equivalence only in some sense. The agreement of behavior on a finite number of inputs provides valuable evidence but does not prove equivalence on all inputs.

V. RESULTS

➢ *Aggregate Held-Out Results*

As per the presented ablation file, testing was conducted on 60 held-out transformed samples (base programs P086P100). These numbers appear in Table I, which summarizes the metrics measured for all configurations. All five lines have the same values. Moreover, the baseline appears to be set as the rule-based approach with pretrained weights not loaded. Thus, one can state that the same results are a flaw of the experiment rather than proof that all components perform equally well.

Table 2 Reported Metrics by Simulated Obfuscation Type

Technique	N	Syntax	AST	CFG	CodeBLEU
AST transform	15	100%	0.997	1.000	0.530
Tokenize transform	15	100%	0.818	1.000	0.735
Custom flattening	15	100%	0.231	0.642	0.124
PyArmor simulation	15	100%	0.382	0.694	0.133

The exact match is 0% for all configurations. This is a crucial finding as the system generated syntactically parseable outputs, none of which matched the normalized reference string. The mean token F1 score of 0.4286 and CodeBLEU of 0.3807 show partial matching, not reconstruction, in most instances. The mean AST similarity of 0.6066 shows moderate structural similarity on average, whereas the CFG similarity is 0.834. These two scores can be different since the flow at the graph level can be similar despite the difference in the actual statements and expressions.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov1219

➢ *Results by Transformation Type*

The provided per-technique table shows varying levels of similarity in terms of the structure and lexicon in the four simulated transformations. Table II shows the provided information regarding the full-pipeline line. All datasets consist of 15 samples and 100% syntax validity. The most similar in terms of ASTs is the AST transformation technique, whereas the custom flattening and the PyArmor-simulated transformation show lower similarity in ASTs and CodeBLEU. This is expected as dispatcher and wrapping transformations generate more divergence from the original program structure in terms of syntax. This does not imply any kind of performance on real samples.

This includes the table per-technique and the aggregate table which must be interpreted only as a summary of the provided artifacts, and not as an assertion of independent replication of performance. For one, the per-technique values do not address the same-configuration problem in the ablation artifact. Prior to making any assertion at the component level, the experiment should be redone with verified model checkpoints and a module audit for each configuration.

➢ *Behavioral Comparison and Confidence Calibration*

The total artifacts report a behavioral match of 0.95. It should be noted that this does not mean equivalence of the semantics of all reconstructed programs but just agreement on the limited set of test examples provided by the supplied harness. The confidence calibration artifact reports a Brier score of 0.1128 and expected calibration error (ECE) of 0.0942 for 60 test examples. It was trained on the validation set and evaluated on the test set based on metadata.

However, these values should be taken with a grain of salt. The test set is small, and there is uneven binning in the reliability statistics, with even some empty high confidence bins. The correctness metric comprises a measure of syntactic correctness and behavioral agreement in the test harness context, but it does not equate to semantic equivalence.

These metrics serve well as initial data, yet further evidence is required for more substantial conclusions.

VI. DISCUSSIONS

The current results illustrate why multiple metrics are required for reconstruction evaluation. The parser will accept the candidate even if it uses a different logic. The CFG similarity will still be high if the general branch structure is preserved, while the token and AST similarities will point out the difference in the statements and expressions. But on the other hand, two semantically equivalent paraphrases can differ on the surface level of text. Various measures give more information than just one measure.

The design of the system also focuses on graceful fallbacks. It is important for an operational analysis tool, where the absence of a model checkpoint should not prevent parsing, feature extraction, and deterministic rules from being used. However, it is necessary to make the fallback explicit in reports and experiment logs. Different ablation configurations that use the same fallback engine will not allow estimating the contribution of transformer inference, AST conditioning, CFG conditioning, and repair.

The synthetic benchmark is suitable for repeat testing, but it is constrained by its lack of diversity. Real obfuscators may perform transformations, rely on runtime conditions, make environment checks, or package the code in an unorthodox format such as not standard Python. Current data set does not generalize obfuscation detection for such cases. It also does not prove that an obfuscation detector can reliably separate malware from legitimate software. Detection of encoding or dynamic calls is merely a clue to further testing.

A. Limitations and Threats to Validity

➢ *Internal Validity*

The ablation metrics being identical, and the use of rulebased fall-back are the main concerns regarding internal validity. This implies that the experiment is not isolating the effects of the model and structure that have been configured. Smoke-test training was done using just two training samples and one validation sample, and the results from the validation phase were not promising at all. A proper training needs to be done with the correct data loading and inference.

➢ *External Validity*

These tests will include benign synthetic programs as well as four transformation categories. These results will not extend to malware developed by other developers, packed programs, unknown transformations in obfuscation, and those that contain a lot of generated code at runtime. More datasets need to be collected using proper authorizations and protection protocols. Diverse sources of the codes must be considered. There should also be a distinction between synthesized and independent samples.

➢ *Metric Validity*

Token and code similarity metrics are proxies for correctness. AST and CFG similarity are dependent on their normalization and matching algorithms. Behavioral testing is limited to certain input samples. A good score in one metric does not mean that the recovered code is correct, accurate, or complete. Confidence scores also rely on correctness labels and distributions.

➢ *Safety and Operational Scope*

A static analysis policy decreases the chances of executing potentially malicious code in production, but there are still other attack vectors. Parsers for the input files, libraries, decompression routines, and service interfaces may need to be secured. The deployment needs to apply least privilege principles, use resource limits, validate uploads, use versioning on dependencies, log and isolate the deployment. The behavior test harness must run outside the static analysis path.

B. Future Work

However, there are some actions that need to be taken in order to draw conclusions about the effectiveness of this approach. First of all, it is necessary to train and validate the

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov1219

trained model on the whole training set and ensure that the correct checkpoint is used for the inference stage. Second, instrument the ablation configurations such that the logs show whether AST summaries, CFG summaries, detection, and repair were used. Third, re-run the experiment to ensure that different configurations can yield different outputs when expected. Fourth, use more diverse base programs, show confidence intervals, and use multiple seeds for the experiments. More analysis will help in providing evidence with regard to each of the transformations that take place. The analysis may consist of showing the original fragment, the suggested transformation, the phase or rule/model which causes it, and the outcome of validation. It is better to use conservative transformations when there is not sufficient evidence to make it safe to transform a certain construct. It is important to note that reconstruction and malware classification have to be separated in future evaluations.

VII. CONCLUSION

This paper introduced DISSECT, a modular approach for static obfuscation analysis and source code recovery of Python programs. The provided implementation uses lexical analysis, AST and CFG creation, pattern recognition, strategy selection, rule-based recovery, potential model inference, syntax fixing, semantic signals, confidence measurement, and orchestration in the background. The evaluation artifacts involve an artificial dataset with 400 obfuscated code variants and 60 test cases. Aggregated results include 100% syntax consistency, 0% exact recovery, token F1 score of 0.4286, CodeBLEU score of 0.3807, AST similarity of 0.6066, and CFG similarity of 0.834.

Most importantly, the key lesson is methodological: the current experiment cannot demonstrate the advantage of the suggested neural network or structure-sensitive features since all ablation rows are the same and the recorded evaluation was performed with the help of the rule-based approach. Confidence scores are provisional and can be evaluated only on the provided synthetic benchmark and its behavioral tests. DISSECT may be viewed as a modular research tool that allows further investigation and cannot be considered as a validated tool for real malware deobfuscation. Model inference verification, improved datasets, proper ablation evaluation, and independent evaluation are crucial next steps.

REFERENCES

[1.] MITRE, “MITRE ATT&CK: Enterprise Matrix.” [Online]. Available: https://attack.mitre.org/. Accessed: Oct. 22, 2026.

[2.] Python Software Foundation, “ast—Abstract Syntax Trees,” *Python 3 Documentation*. [Online]. Available: https://docs.python.org/3/library/ast.html. Accessed: Oct. 22, 2026.

[3.] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proc. EMNLP*, 2021, pp. 8696–8708.

[4.] S. Ren, D. Guo, S. Lu, L. Zhou, N. Tang, M. Zhou, and S. Liu, “CodeBLEU: A method for automatic evaluation of code synthesis,” arXiv preprint arXiv:2009.10297, 2020.

[5.] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, “Syntia: Synthesizing the semantics of obfuscated code,” in *Proc. USENIX Security Symposium*, 2017, pp. 643–659.