Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Automated Test Case Generation for Airborne Software Using a VAE–Transformer Model

Wenguang Xie
Northwestern Polytechnical University
Xiaoxue Zeng
cherzeng1012@163.com
Civil Aviation University of China
Jie Bai
Civil Aviation University of China
Kenian Wang
Civil Aviation University of China
Kun Yang
Civil Aviation University of China

Research Article

Keywords: airborne software testing, test case generation, variational autoencoder, fuzzing, Transformer, deep learning

Posted Date: October 26th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11294676/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Automated Test Case Generation for Airborne Software Using a VAE–Transformer Model

Abstract

Airborne software is safety-critical and typically features strict input formats and complex operational states. A central testing challenge is to achieve adequate coverage of deep state-dependent execution paths. Conventional coverage-guided fuzzing relies on random byte-level mutations, which can corrupt input structures and fail to capture state dependencies in event sequences, making it difficult to generate high-quality test cases that reach deep execution paths. To address these limitations, this paper proposes a VAE–Transformer-based method for automated test case generation for airborne software. First, a structured representation is developed for airborne state-machine inputs: fixed-length byte sequences are represented as an $m \times n$ event–field matrix. Based on an analysis of flight-control state transitions, key state attributes, including system mode, arming state, and failsafe level, are explicitly encoded into an s-dimensional operational-state semantic vector, enabling the joint representation of input structure and state context. Second, a variational autoencoder learns the latent distribution of high-value test cases subject to state-semantic constraints, and reparameterized sampling provides diverse candidates under the same conditioning vector. An event-aware gated fusion mechanism adaptively balances the contributions of the conditioning semantic vector and the stochastic latent variable. A Transformer encoder then models bidirectional dependencies in the event sequence and generates diverse, complete test cases in parallel to explore additional execution paths. Experiments on several representative airborne software targets show that the proposed method outperforms the comparison methods. For the representative flight-control command program `commander`, edge, line, and branch coverage improve by approximately 4.9, 5.1, and 4.6 percentage points, respectively, over the AFL++ baseline , while the execution budget required to reach the target coverage decreases by approximately 9.6%. The average execution budget across six target programs decreases by approximately 9.4%. These results indicate that the proposed method improves both coverage and efficiency in airborne software fuzzing and offers a feasible approach to automated testing of high-assurance software.

Keywords: airborne software testing, test case generation, variational autoencoder, fuzzing, Transformer, deep learning

1 Introduction

As avionics systems become increasingly complex [1] and airborne software assumes a larger share of system functionality [2], software has become essential to flight control, state monitoring, mission management, and other critical functions. Because airborne software is typically assigned a high design assurance level [3], testing must establish functional correctness [4] and satisfy stringent adequacy requirements for structural coverage, exception handling, and state transitions. Conventional test case design and generation, however, rely heavily on manual analysis of software requirements. For complex control logic and interactions involving multiple events, this approach often suffers from low generation efficiency, insufficient coverage, and limited exploration of deep state-dependent paths. Coverage-guided fuzzing continuously selects high-value inputs using program execution feedback and has become an important technique for automated test case generation and software robustness testing. Its ability to explore deep state-dependent paths is commonly assessed using metrics such as edge and branch coverage [5]. Greybox fuzzers, including AFL and AFL++, repeatedly mutate existing seeds based on coverage feedback [6] to discover new program paths. However, airborne software typically imposes strict input formats and exhibits complex operational states. Its critical functions often require multiple events to occur jointly under specific state conditions, making deep paths particularly difficult to cover. For the structured inputs commonly used in airborne software, general-purpose random mutations can disrupt field boundaries, event order, and state constraints, causing many inputs to be rejected during parsing or state validation. In flight-control state-machine programs in particular [7], critical paths often depend on combinations of event timing, field values, and operational states. Conventional byte-level mutation alone therefore struggles to produce high-quality test cases that preserve structural validity and satisfy state-semantic constraints. In coverage-guided fuzzing, the implementation of state transitions involves program branches and control-flow edges. Reaching a deep branch guarded by a state condition indicates that the corresponding condition has been satisfied and that the associated state-dependent path has been executed [4]. In recent years, deep learning [8] has increasingly been used to support input generation and optimize fuzzing. Previous studies [9–11] suggest that generative models can learn the structural distributions of inputs and improve sample diversity to some extent. Nevertheless, most learning-based methods focus on textual protocols or file formats. Relatively little attention has been paid to the fixed-length binary inputs of airborne software state machines, which have explicit field boundaries and strong state dependencies. To address these challenges, this paper proposes a VAE–Transformer-based method for automated airborne software test case generation. The method represents fixed-length binary inputs as an $m \times n$ event–field matrix and uses runtime information from the target program to construct an s-dimensional state-semantic conditioning vector. A VAE learns the latent distribution of high-value inputs, while a gated fusion mechanism adaptively combines the conditioning semantics with latent representations. A Transformer models bidirectional dependencies among events and fields to generate test cases. After postprocessing, these cases serve as the initial corpus for AFL++, improving the efficiency of path exploration. The two principal contributions are as follows:

1. **Semantic-extraction-based VAE latent modeling.** To address the lack of structural semantic representations for raw binary inputs and the difficulty of modeling the distribution of high-value test cases, binary inputs are reorganized into an event–field matrix that preserves event semantics. An s-dimensional operational-state semantic vector is extracted as an explicit condition. A conditional VAE maps the state semantics to a latent distribution, and reparameterized sampling provides a continuous, controlled source of diversity under the same state condition. This design explicitly separates state-semantic constraints from the source of sample diversity.
2. **Event-aware gated Transformer generation of structured test cases.** To preserve bidirectional relationships within event sequences and dependencies on operational states, an event-aware gating mechanism adaptively balances the conditioning semantic vector and stochastic latent vector at each event position. A Transformer encoder models the contextual relationships of the fused event representations, supporting both structural validity and path exploration in the generated samples.

2 Related Work

2.1 Automated Generation Using Conventional Algorithms

As greybox fuzzing has matured, coverage-guided methods represented by AFL and AFL++ have become a mainstream approach. They combine lightweight instrumentation with a closed loop of seed selection, input mutation, target execution, coverage feedback, and retention of high-value seeds. Improvements have primarily addressed four aspects: seed scheduling, directed path exploration, constraint handling, and structure preservation. For seed scheduling, Böhme et al. [6] modeled execution paths as a Markov chain and allocated mutation energy according to path visitation frequency to accelerate exploration of infrequently executed code. CollAFL [12] improved path discrimination by eliminating coverage-bitmap collisions, while FairFuzz [13] prioritized mutations of seeds that trigger rare branches. More recently, DiPri [14], a distance-based seed prioritization method, further improved scheduling efficiency. For directed path exploration, AFLGo [15] used the distance from program locations to target sites and a simulated annealing strategy to concentrate mutation budgets on seeds closer to those targets. For constraint handling, Steelix [16] and REDQUEEN [17] exploited correspondences between input bytes and comparison instructions to repair magic-byte and checksum constraints. Angora [18] combined byte-level taint tracking with gradient descent to solve branch conditions. QSYM [19] and T-Fuzz [20] overcame complex checks using lightweight concolic execution and program transformation, respectively. For structure preservation, Superion [21] performed subtree-level mutations on abstract syntax trees to maintain the syntactic validity of structured inputs. FormatFuzzer [22] generated and mutated binary-format inputs using format templates to preserve structural validity. For stateful programs, StateAFL [23] and NSFuzz [24] inferred runtime states and constructed state-aware feedback to improve the exploration of state-dependent paths. Overall, classical approaches face inherent

limitations associated with path explosion and expensive constraint solving. Moreover, coverage-guided fuzzers still rely heavily on random operations such as bit flips and byte substitutions. For structured binary inputs with explicit field boundaries, event ordering, and state constraints, many mutated inputs fail during parsing or state validation, limiting access to deep paths that depend on combinations of events in airborne state-machine programs.

2.2 Deep-Learning-Assisted Fuzzing Test Case Generation

Deep learning has increasingly been incorporated into fuzzing, primarily through input generation and input-value prediction [25]. Input-generation methods use neural networks to learn input-format distributions and improve parsability and structural validity. Learn&Fuzz [10] was an early implementation of this idea: it treated PDF objects as character sequences and trained a character-level recurrent neural network to guide structured input generation, demonstrating the feasibility of learning-based generation on a PDF parser. SeqFuzzer [26] subsequently used sequence-to-sequence modeling and LSTM networks to learn message structures and temporal patterns in stateful protocols. GANFuzz [9] learned protocol syntax from industrial protocol messages using generative adversarial networks. IUST-DeepFuzz [25] extended recurrent neural networks to complex file formats containing nontextual data. For C compilers, DeepFuzz [27] used a sequence-to-sequence generative model to continuously produce syntactically valid C programs for fuzzing compilers such as GCC and Clang, demonstrating the effectiveness of learning-based generation in compiler testing. COMFORT [28] used GPT-2 to generate JavaScript test programs and combined differential and conformance testing with the ECMAScript specification to reveal deviations between engine implementations and the standard. Input-value prediction methods estimate the potential testing value of inputs to optimize fuzzing execution. FuzzGuard [29] used a convolutional neural network to predict target reachability and filter unreachable seeds in directed greybox fuzzing. Rajpal et al. [30] used an LSTM to predict the value distribution of input bytes from historical mutations and coverage information, guiding subsequent mutations. MEUZZ [31] learned seed-scheduling decisions from reachability and dynamic-analysis features. Wang et al. [32] applied reinforcement learning to adaptive hierarchical coverage scheduling, while Huang et al. [33] used Monte Carlo planning to balance scheduling benefits against costs. These studies suggest that deep learning can partly mitigate the limitations of random mutation for structured inputs. However, existing research primarily concerns textual protocols, file parsers, and general-purpose programs. Systematic investigation remains limited into generative models that preserve both structural validity and state semantics for the fixed-length, explicitly structured, strongly state-dependent binary inputs of airborne software state machines.

3 Preliminaries

3.1 Variational Autoencoder

A variational autoencoder (VAE) is a probabilistic latent-variable generative model that represents the underlying distribution of observed samples in a continuous latent space. Unlike a conventional autoencoder, which maps an input to a single deterministic latent vector, a VAE encodes an input as a probability distribution over latent variables. To generate a new sample, a latent variable is first sampled from this distribution and then decoded into a sample consistent with the training-data distribution. A VAE therefore supports representation learning while providing a sampleable prior for generation. Let X denote a structured test sample, c the conditioning semantic vector, and z the latent variable. In a conditional VAE, the encoder represents the input and its state condition through the approximate posterior $q_\phi(z \mid X, c)$. This posterior is commonly assumed to be Gaussian:

$$q_\phi(z \mid X, c) = \mathcal{N}\big(z; \mu_\phi(X, c), \mathrm{diag}\big(\sigma_\phi^2(X, c)\big)\big)\,. \tag{1}$$

Here, $\mu_\phi(X, c)$ and $\sigma_\phi(X, c)$ are the mean and standard deviation predicted by the encoder, respectively, and ϕ denotes its parameters. To enable gradient backpropagation through stochastic sampling, the VAE uses the reparameterization trick:

$$z = \mu_\phi(X, c) + \sigma_\phi(X, c) \odot \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, I). \tag{2}$$

Here, ε is noise sampled from the standard normal distribution, and $\odot$ denotes elementwise multiplication. Reparameterization enables end-to-end training while retaining stochastic sampling. The VAE objective generally comprises a reconstruction term and a Kullback–Leibler (KL) divergence regularizer:

$$\mathcal{L}_{\mathrm{VAE}} = -\mathbb{E}_{z \sim q_\phi(z \mid X, c)}[\log p_\phi(X \mid z, c)] + D_{\mathrm{KL}}(q_\phi(z \mid X, c) \,\|\, p(z))\,. \tag{3}$$

The first term penalizes differences between the reconstructed output and the original sample; the second constrains the approximate posterior $q_\phi(z \mid X, c)$ to remain close to the prior $p(z)$. Together, they enable the model to reconstruct observed samples and learn a continuous, smooth, sampleable latent space. In this study, the VAE does not directly generate complete byte streams. Instead, it learns the latent distribution of high-value test cases under state conditions. For structured test samples with similar operational-state semantics, it provides candidate latent representations that combine randomness with distributional consistency. These representations support the subsequent Transformer-based generation of event-level field sequences, helping to maintain sample diversity and explore potentially high-value paths.In this work, the encoder is conditioned on c only, and X serves as the reconstruction target (Section 4.1.2).

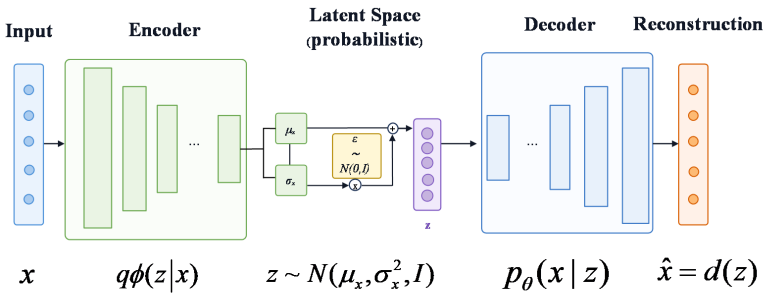

Fig. 1 Architecture of a standard variational autoencoder

3.2 Transformer Encoder and Multi-Head Self-Attention

The Transformer is an attention-based sequence-modeling architecture whose central mechanism directly models dependencies between arbitrary sequence positions through self-attention. Given an input sequence, the model constructs query, key, and value matrices, denoted by Q, K, and V, respectively, and computes relationships between positions using scaled dot-product attention. Compared with RNNs and LSTMs, Transformers offer more stable modeling of long-range dependencies and greater parallelism. They are therefore widely used in text generation, structured sequence modeling, and multimodal learning. Self-attention computes similarities between query and key vectors to determine how much information each position receives from other positions, and then forms a weighted sum of the value vectors. This allows the model to determine dynamically which parts of the sequence matter to a given element. For airborne state-machine inputs, an event can therefore depend directly on distant mode changes, fault injections, or recovery conditions, rather than only on neighboring events, enabling the model to capture logical relationships across events.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathsf{T}}}{\sqrt{d_k}}\right) V. \tag{4}$$

Multi-head self-attention extends single-head attention by projecting the input into multiple representation subspaces, computing attention independently in each subspace, and concatenating and projecting the results back to a common dimension. This enables the model to examine the sequence from several representational perspectives simultaneously. For example, one head may focus on event order, another on co-occurring fault flags, and another on compatibility between mode changes and arming states. Multi-head attention thus generally offers greater representational capacity

than a single attention head. It is defined as

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W^O, \tag{5}$$

where the ith head is

$$\mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V). \tag{6}$$

Multi-head self-attention is only one sublayer of a complete Transformer encoder block. A full block also includes a position-wise feed-forward network, residual connections, and layer normalization. These components jointly support contextual modeling, representation transformation, and stable optimization. The feed-forward network applies a nonlinear transformation at each position to increase representational capacity. Residual connections preserve the original information and alleviate gradient degradation in deep networks, while layer normalization stabilizes training and reduces numerical variation across feature dimensions. Stacking multiple encoder blocks yields increasingly abstract and global sequence representations. The feed-forward network is commonly expressed as

$$\mathrm{FFN}(x) = W_2\sigma(W_1x + b_1) + b_2. \tag{7}$$

The encoder sublayers with residual connections and layer normalization can be written as

$$\widetilde{x} = \mathrm{LayerNorm}(x + \mathrm{MultiHead}(x, x, x)), \tag{8}$$

$$y = \mathrm{LayerNorm}(\widetilde{x} + \mathrm{FFN}(\widetilde{x})). \tag{9}$$

In the present task, the input is an event-level hidden representation mapped from the conditioning semantic vector, rather than a sequence of natural-language tokens. The conditioning representation is first mapped into an event-level latent space. Multi-head self-attention then models global dependencies among the m events, followed by feed-forward transformations and layer normalization. The resulting output is a field-level representation with the same structure as the original test case.

4 VAE–Transformer Framework for Automated Test Case Generation

To address strict input formats, complex state constraints, and the scarcity of high-value test samples in airborne software, this paper proposes a VAE–Transformer-based generation framework. On the modeling side, semantic-extraction-based VAE latent modeling reorganizes fixed-length byte inputs into an $m \times n$ event–field matrix and extracts an s-dimensional operational-state semantic vector c by executing historical seeds. The VAE learns a latent distribution and samples z through reparameterization, providing a source of diversity for seed generation under the same state semantics. On the generation side, event-aware gated Transformer generation adaptively balances the contributions of c and z at each event position. A Transformer encoder models bidirectional long-range dependencies among events, and the fused representation is

converted into a byte sequence in parallel, supporting structural validity, state consistency, and path exploration. The framework comprises four stages: data processing, VAE latent modeling, event-aware gated Transformer generation, and independent evaluation using AFL++. AFL++ uses the generated test cases as its initial corpus for coverage-guided fuzzing; its coverage feedback does not update the generative model parameters.

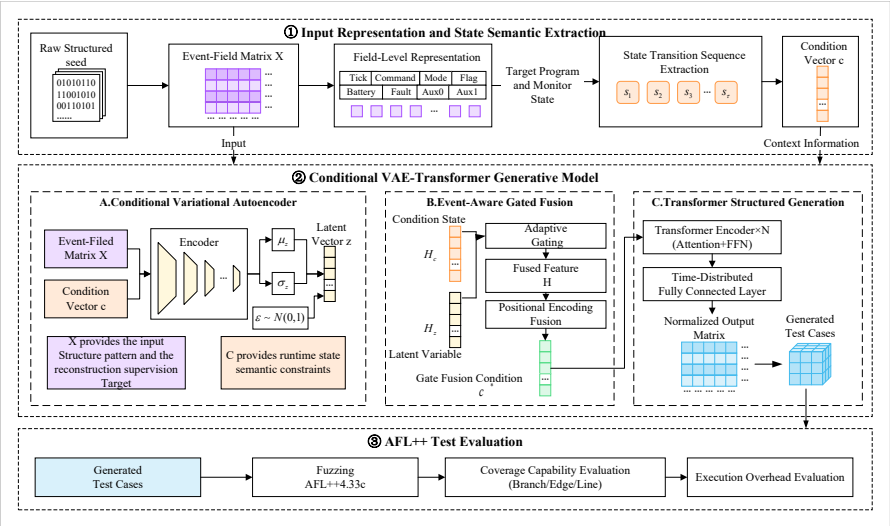

Fig. 2 VAE–Transformer framework for automated test case generation

1. **Input representation and state-semantic extraction:** Original structured seeds → Configurable event–field matrix X → Field-level representation (fields of each event: Tick, command, mode, flags, battery level, fault, auxiliary 0, auxiliary 1) → Execute the seeds on the target program and monitor its states → State-transition sequence extraction $(s_1, s_2, s_3, \ldots, s_T)$ → Conditioning vector c. The two downward labels are "Input-pattern information" and "Operational-state context information."
2. **Conditional VAE–Transformer generative model: A. Conditional variational autoencoder (VAE):** Configurable event–field matrix X; Conditioning vector c; Encoder; μ_z; σ_z; $\varepsilon \sim \mathcal{N}(0,1)$; Latent variable z; "X provides the input-structure pattern and reconstruction target"; "c provides operational-state semantic constraints." **B. Event-aware gated fusion:** Conditional-state representation H_c; Latent representation H_z; Adaptive gate-weight generation; Fused feature representation H; Addition of positional encoding; Gated conditioning representation c^*. **C. Transformer-based structured generation:** Transformer encoder block $\times$ N (multi-head attention + feed-forward network) → Time-distributed fully connected layer (8 dimensions, sigmoid) → Normalized output matrix $\widehat{X}$ → Generated structured test cases.

3. **AFL++-based testing and evaluation:** Generated structured test cases → AFL++ fuzzing → Coverage evaluation (branch/edge/line coverage) → Execution-budget evaluation.

4.1 Semantic-Extraction-Based VAE Latent Modeling

Conventional fuzzing treats test inputs as continuous byte streams, making it difficult to preserve event boundaries, field positions, and operational-state constraints explicitly. Random byte mutations can also damage input structure and distort state semantics. The proposed conditional latent modeling method uses a conditioning vector c to impose operational-state semantic constraints on generation, supporting structural plausibility and state consistency. A latent variable z simultaneously provides stochastic variation under the same condition, improving candidate diversity and path exploration without discarding the semantic constraints altogether.

4.1.1 Structured Input Representation and Operational-State Semantic Extraction

Each fixed-length binary input is first reorganized into an event-level field matrix, with rows corresponding to event steps and columns to field dimensions. This representation preserves local field semantics and global event order, providing an explicit structural basis for subsequent attention modeling:

$$X = [e_1, e_2, \ldots, e_T] \in \{0, 1, \ldots, 255\}^{T \times F}, \qquad T = m, \quad F = n. \tag{10}$$

In addition to input structure, operational-state semantics determine whether an input is effective. Each test case is therefore executed on the target program to extract a runtime-state summary $S(X)$, including the final mode, arming state, failsafe level, rejection statistics, and several bitmask features. These features are concatenated in a fixed order and normalized to obtain an s-dimensional conditioning vector c. Unlike a representation derived solely from input content, this vector explicitly describes the state semantics observed after execution, integrating input structure and runtime context into a common generation framework. Let $X \in \mathbb{R}^{m \times n}$, where m is the number of event steps and n is the number of fields per event. The conditioning vector $c \in \mathbb{R}^{13}$ combines discrete semantic attributes, continuous count features, and bitmask features to characterize the program state and contextual constraints. Training uses paired samples (X, c), so generation is guided by explicit state semantics rather than by X alone.

4.1.2 Modeling the Conditioning Vector and Latent Variable

Traditional fuzzing obtains diversity through random byte mutation, which readily disrupts field boundaries and state dependencies in structured inputs. The proposed approach instead uses conditional VAE latent modeling to map the high-dimensional, discrete input space to a low-dimensional, continuous latent space and sample under the guidance of c. Structural diversity is obtained through smooth interpolation and

stochastic perturbations in the continuous latent space, rather than through unconstrained bit flips. The latent encoder takes the operational-state semantic conditioning vector c as input and learns a conditional distribution over z. After concatenation or joint mapping, the conditioning information is passed to the conditional encoder to predict the distribution parameters μ and $\log \sigma^2$. The resulting conditional posterior $q_\phi(z \mid c)$ defines the latent space of high-value candidates sharing the same state semantics:

$$q_\phi(z \mid c) = \mathcal{N}\big(\mu_\phi(c), \operatorname{diag}\big(\sigma_\phi^2(c)\big)\big)\,. \tag{11}$$

The latent variable z is then obtained by reparameterized sampling. The conditioning vector c specifies the state-semantic constraints, while z introduces continuous stochastic perturbations under these constraints, allowing candidates with different structural details to be generated for the same state semantics:

$$z = \mu_z + \exp\left(\frac{1}{2}\log \sigma_z^2\right) \odot \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, I). \tag{12}$$

At inference time, the original input is no longer required; the latent variable is sampled directly from the standard normal distribution:

$$z \sim \mathcal{N}(0, I). \tag{13}$$

The original seed X is not fed directly into the latent encoder. Instead, it serves as the supervision target for the Transformer encoder output $\widetilde{X}$. Through the reconstruction loss, structural information is backpropagated to the conditional latent encoder, gated fusion module, and main generator, enabling end-to-end joint optimization. To align the conditional latent distribution learned during training with standard normal sampling at inference, the latent space is regularized using

$$D_{\mathrm{KL}}(q_\phi(z \mid c) \,\|\, \mathcal{N}(0, I))\,. \tag{14}$$

This regularization encourages a continuous, sampleable latent space. At inference time, $z \sim \mathcal{N}(0, I)$ and the target condition c jointly drive generation. In contrast to discrete random flips in the raw byte space of AFL++, the proposed method introduces randomness in a continuous latent space and generates candidates under state conditions. It thus replaces unconstrained byte mutation with condition-guided continuous latent variation, expanding structural diversity while retaining state-semantic constraints. This design explicitly separates semantic constraints from the source of diversity: c specifies the state-semantic boundaries that generated outputs should satisfy, while z provides diverse candidates under the same condition. This separation helps avoid the loss of structural integrity associated with purely random mutation.

4.2 Structured Generation Using an Event-Aware Gated Transformer Encoder

A linear combination of c and z cannot readily balance state constraints against stochastic diversity. The proposed event-aware gated Transformer method therefore

adaptively controls the relative contributions of the state-semantic conditioning vector and stochastic latent variable at each event position. This provides dynamic coordination between constraint satisfaction and diversity. A Transformer encoder then models bidirectional long-range dependencies among events to generate complete test cases with coherent global structure.

4.3 Event-Aware Gated Fusion

The conditioning vector c and latent variable z are first projected into a common hidden space through separate learnable linear mappings and reshaped into initial representations for the m event positions:

$$H_c = \text{Reshape}(W_c c + b_c), \tag{15}$$

$$H_z = \text{Reshape}(W_z z + b_z). \tag{16}$$

Here, H_c is the deterministic event-level semantic representation derived from the operational-state condition, and H_z is the event-level latent representation derived from the stochastic latent variable. Both have the same event-sequence length and hidden dimension, enabling position-wise fusion. The two representations are then passed jointly to the gating unit to compute adaptive weights:

$$G = \sigma(W_g[H_c; H_z] + b_g)\,. \tag{17}$$

Here, $[H_c; H_z]$ denotes concatenation, $\sigma(\cdot)$ is the sigmoid activation function, and each element of G lies in $[0, 1]$. The fused representation is

$$H_{\text{fuse}} = G \odot H_c + (1 - G) \odot H_z, \tag{18}$$

where $\odot$ denotes elementwise multiplication. The gate adaptively balances state-condition information and stochastic latent information according to the event position and hidden features. Larger gate values retain more of the state semantics supplied by the conditioning vector, whereas smaller values allow stochastic latent variation to exert a greater influence on generation. Functionally, the conditioning branch imposes state-semantic constraints, the latent branch supplies random variation, and the gate coordinates their contributions at the event level. Compared with fixed addition or simple concatenation, event-aware gating provides position-specific fusion and reduces the risk of an imbalance between semantic conditioning and stochastic variation. Positional encoding P is added to the fused event representations to preserve the order of the m events:

$$H^{(0)} = H_{\text{fuse}} + P. \tag{19}$$

4.3.1 Structured Generation with the Transformer Encoder

The representation $H^{(0)}$ is passed to the Transformer encoder backbone. Multi-head self-attention allows each event position to attend to every other position in the

complete event sequence. In an attention layer, the computations are

$$Q = H^{(l-1)}W_Q, \tag{20}$$

$$K = H^{(l-1)}W_k, \tag{21}$$

$$V = H^{(l-1)}W_v, \tag{22}$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathsf{T}}}{\sqrt{d_k}}\right)V. \tag{23}$$

Multi-head attention models relationships among events in parallel across different projection subspaces, capturing long-range dependencies. By incorporating the context of subsequent events, it provides bidirectional structural modeling of the entire test-case sequence. Each Transformer encoder layer comprises multi-head self-attention, residual connections, layer normalization, and a feed-forward network:

$$\widetilde{H}^{(l)} = \text{LayerNorm}\Big(H^{(l-1)} + \text{MultiHeadAttention}(H^{(l-1)})\Big), \tag{24}$$

$$H^{(l)} = \text{LayerNorm}\Big(\widetilde{H}^{(l)} + \text{FFN}(\widetilde{H}^{(l)})\Big). \tag{25}$$

Multi-head self-attention aggregates contextual information across event positions, while the feed-forward network performs a nonlinear transformation of each event representation. After multiple layers, the model produces a high-level structured representation $H^{(L)}$ that incorporates global event dependencies and local field semantics. Finally, a fully connected output mapping shared across events converts each event representation into a field-prediction vector:

$$\widehat{X} = \sigma(H^{(L)}W_o + b_o). \tag{26}$$

The implementation uses `TimeDistributed(Dense(8, sigmoid))`, applying the same eight-dimensional fully connected output layer to all m event positions and generating the complete normalized $m \times n$ matrix in parallel. Unlike byte-by-byte autoregressive generation, the model does not generate each subsequent byte from previous byte predictions. Instead, it outputs all fixed fields simultaneously using the full event context, improving the efficiency of structured generation. The normalized matrix $\widehat{X} \in [0, 1]^{m \times n}$ is denormalized, rounded, and clipped to produce a discrete matrix within the valid byte range:

$$X_{\text{byte}} = \text{Clip}\Big(\text{Round}(255\widehat{X}), 0, 255\Big). \tag{27}$$

The byte matrix is then flattened in the original event and field order to reconstruct a fixed-length binary input of $m \times n$ bytes. This input becomes a candidate initial seed for subsequent AFL++ fuzzing. During training, the true matrix X is not a direct

input to either the conditional latent encoder or the Transformer encoder; it serves as the supervision target. Minimizing the discrepancy between $\widehat{X}$ and X backpropagates structural information through the output layer, Transformer encoder, gated fusion module, and reparameterization path, enabling end-to-end joint optimization. The overall training objective combines reconstruction loss and KL regularization:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{rec}} + \beta \mathcal{L}_{\text{KL}}. \tag{28}$$

Here, $\mathcal{L}_{\text{rec}}$ penalizes differences between $\widehat{X}$ and X, $\mathcal{L}_{\text{KL}}$ regularizes the conditional latent posterior, and β balances the two terms.

1) Reconstruction loss. The generated normalized matrix $\widehat{X} \in [0,1]^{m \times n}$ is compared elementwise with the true normalized matrix X using mean squared error:

$$\mathcal{L}_{\text{rec}} = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (\widehat{X}_{ij} - X_{ij})^2. \tag{29}$$

Here, m is the event-sequence length, n is the number of fields per event, and X_{ij} and $\widehat{X}_{ij}$ are the true and generated normalized values of field j in event i, respectively. Since all fields are normalized to $[0,1]$, elementwise mean squared error places equal-scale constraints on the protocol fields and prevents fields with larger numerical ranges from dominating the gradients. Compared with byte-level cross-entropy supervision, matrix-level reconstruction loss also matches the simultaneous generation of all fields, allowing the supervision signal to act on every event position at once.

2) KL regularization. The conditional latent encoder predicts the posterior $q_\phi(z \mid c) = \mathcal{N}(\mu, \text{diag}(\sigma^2))$. The KL term constrains this posterior to remain close to $\mathcal{N}(0, I)$, reducing the distribution mismatch between reparameterized sampling during training and standard normal sampling during inference:

$$\begin{aligned} \mathcal{L}_{\text{KL}} &= D_{\text{KL}}\big(\mathcal{N}(z; \mu, \text{diag}(\sigma^2)) \,\|\, \mathcal{N}(0, I)\big) \\ &= \frac{1}{2} \sum_{k=1}^{d} \left(\mu_k^2 + \sigma_k^2 - \ln \sigma_k^2 - 1\right). \end{aligned} \tag{30}$$

Here, μ_k and σ_k^2 are the mean and variance estimated for latent dimension k. During training, the model samples from the posterior using $z = \mu + \sigma \odot \varepsilon$, with $\varepsilon \sim \mathcal{N}(0, I)$; during inference, it samples directly from the standard normal prior. Without KL regularization, the posterior may drift away from the prior, causing inference-time samples to fall in regions not encountered during training and reducing generation quality. Excessively strong KL regularization, however, can collapse the posterior toward the prior and remove conditioning information from the latent variable. The coefficient β must balance these competing effects. The gated fusion module requires no independent supervision. Its parameters share the same loss function $\mathcal{L}$ as the conditional latent encoder, Transformer encoder, and output layer. Reconstruction gradients propagate from the output layer through the Transformer encoder and gated fusion module, then through the reparameterization path to the mean and variance

branches of the conditional latent encoder. KL gradients act directly on the encoder's distribution parameters. These two gradient paths jointly update the model, allowing latent representations, gate weights, and structured generation to be optimized under one objective without staged training or an additional alignment loss. After training, the model parameters remain fixed. Generated candidates undergo byte reconstruction and validity filtering before being supplied to AFL++ as the initial corpus for coverage evaluation.

5 Experimental Analysis

All experiments were conducted on a Dell PowerEdge T550 server running Ubuntu 22.04 LTS. The hardware comprised two Intel Xeon Silver 4309Y processors with 16 cores in total at 2.80 GHz, two NVIDIA Tesla T4 GPUs, 32 GB of memory, and 3.6 TB of disk storage. The software environment used AFL++ 4.33c for fuzzing, GCC 9.5.0 for program compilation, and Python 3.10 for experimental scripts. We compared three test case generation approaches: conventional coverage-guided fuzzing (denoted as AFL),GAN-assisted generation, and VAE–Transformer-assisted generation. The AFL baseline uses the original seed corpus without deep-learning-based seed generation. AFL++ version 4.33c serves as the underlying fuzzing engine for all three approaches.

5.1 Experimental Setup

To preserve temporal structure and field semantics in command sequences while providing fixed-dimensional inputs for stable training, all inputs were set to 128 bytes and mapped to a 16×8 event–field matrix. The 16 rows represent event steps, and the eight columns represent the fields associated with each event. The VAE conditioning vector was set to 13 dimensions to describe program states and contextual constraints. The comparison GAN retained the coverage-bitmap encoding used by its original method [34]; the two models used the same source of conditioning information but different encodings. These settings provided a common experimental basis for evaluating generated-sample quality, path coverage, and model performance. Table 1 lists the core hyperparameters used during training and inference. Six key control-logic modules from the PX4 flight-control system were selected: `commander` (flight-state management and mode switching), `mission_manager` (mission-execution control), `health_monitor` (system health monitoring), `command_router` (routing and dispatch of commands from multiple sources), `failsafe_manager` (safety protection and failure handling), and `recovery_supervisor` (post-fault recovery supervision and safe fallback). Together, these modules represent critical airborne software functions, including state control, mission management, health monitoring, safety protection, and fault recovery. The comparison included the AFL++ baseline, a conditional generative GAN, and the proposed VAE–Transformer. The evaluation metrics were edges covered (EC), lines covered (LC), branches covered (BC), and the execution budget required to reach the target coverage. The first three quantify coverage, while the execution budget measures testing efficiency. These metrics were used to assess coverage improvements, execution-budget reductions, and consistency across target modules.The execution budget is defined as the cumulative number of executions required for each method

to first reach the final edge coverage achieved by the AFL++ baseline on the same target, as recorded in AFL++'s plot data.

Table 1 Model hyperparameters.

Parameter	Value
Learning rate	0.0008
Latent dimension	8
Backbone dimension	64
Batch size	32
Epochs	40
Number of attention heads	4
Conditioning dimension (`Cond_dim`)	13

5.2 Comparative Experimental Analysis

To evaluate the proposed method for structured generation and coverage-guided fuzzing, AFL and GAN were selected as representative comparison methods. AFL is a classical coverage-guided fuzzer: its seeds are not processed by a generative model, and it explores paths using random mutations and coverage feedback. GAN represents a widely studied deep-learning approach to test case generation, learning the input distribution through adversarial training. Comparison with GAN therefore helps assess improvements in structural modeling and generation quality. Because no single metric can fully characterize testing performance, this section considers both coverage and efficiency, using coverage-growth trends to examine the path-exploration behavior of each method.

Table 2 Coverage and execution efficiency on the `commander` target program.

Metric	AFL	GAN	VAE–Transformer
Edges covered	220	225	232
Branches covered	290	297	305
Lines covered	263	265	278
Execution budget (million executions)	30	28	27

As shown in Table 2, VAE–Transformer consistently performs better in both coverage and testing efficiency on `commander`. It covers 232/246 edges, compared with 225/246 for GAN and 220/246 for AFL. Its line coverage reaches 278/296, covering 15 more lines than AFL, while branch coverage reaches 305/326, also covering 15 more branches. The coverage-growth curves indicate that VAE–Transformer reaches a higher coverage level earlier and has essentially stabilized at its final value by approximately

2 h. This suggests that it uses the testing budget more efficiently to reach high-value code regions. Overall, GAN improves on AFL for some metrics, but the magnitude and consistency of its gains are lower than those of VAE–Transformer. The proposed model shows stronger representation learning, input-structure preservation, and cross-target generalization. These advantages are attributed to the VAE's ability to capture the distribution of high-value test cases in latent space and the Transformer's ability to represent global relationships among fields, events, and states.

5.3 Generalization Across Programs Under Test

To assess generalization across airborne software targets, further comparisons were conducted on `commander`, `mission_manager`, `health_monitor`, `command_router`, `failsafe_manager`, and `recovery_supervisor`, which differ in state complexity and control logic. VAE–Transformer, GAN, and the AFL++ baseline were evaluated in terms of edge coverage, branch coverage, line coverage, and execution budget.

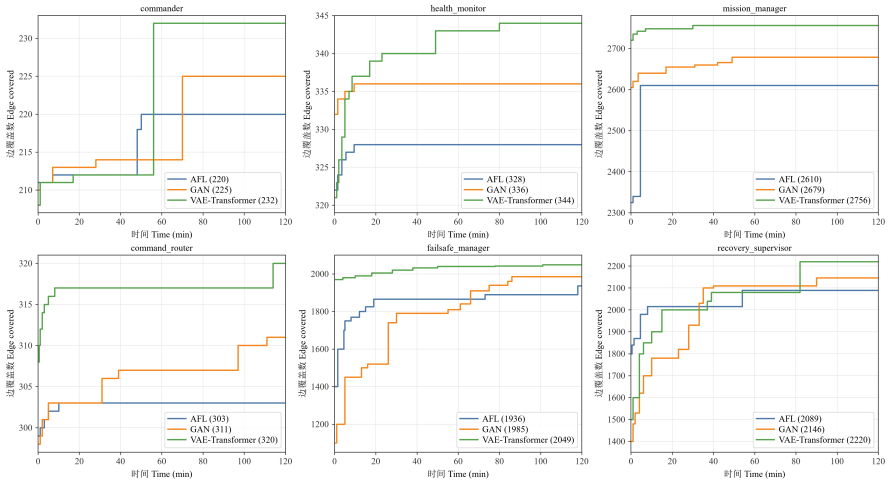

Fig. 3 Edge-coverage growth for the compared models on key PX4 modules
English axis labels: horizontal—Time (min); vertical—Edges covered. Panel titles and model names are unchanged.

As shown in Fig. 3, edge coverage increases gradually for all three methods and approaches a plateau during the later stages of testing. VAE–Transformer achieves the highest final edge count on all six programs. On `mission_manager`, for example, it discovers 2,756 edges, exceeding GAN and AFL by 77 and 146 edges, respectively. On `recovery_supervisor`, it discovers 2,220 edges, exceeding the two baselines by 74 and 131 edges. Across the six programs, AFL, GAN, and VAE–Transformer cover 7,486, 7,682, and 7,921 edges, respectively. The proposed model therefore covers 239 more edges than GAN and 435 more than AFL, corresponding to increases of approximately 3.11% and 5.81%. VAE–Transformer also shows more sustained coverage growth on most programs. For example, on `health_monitor`, `mission_manager`, and

`failsafe_manager`, it continues to discover edges after the growth of AFL or GAN has slowed. Growth patterns nevertheless vary by program: most gains on `commander` occur during the middle of testing, whereas gains on `command_router` are concentrated near the beginning. Thus, the model's benefit extends beyond early exploration speed to its ability to overcome intermediate coverage plateaus.

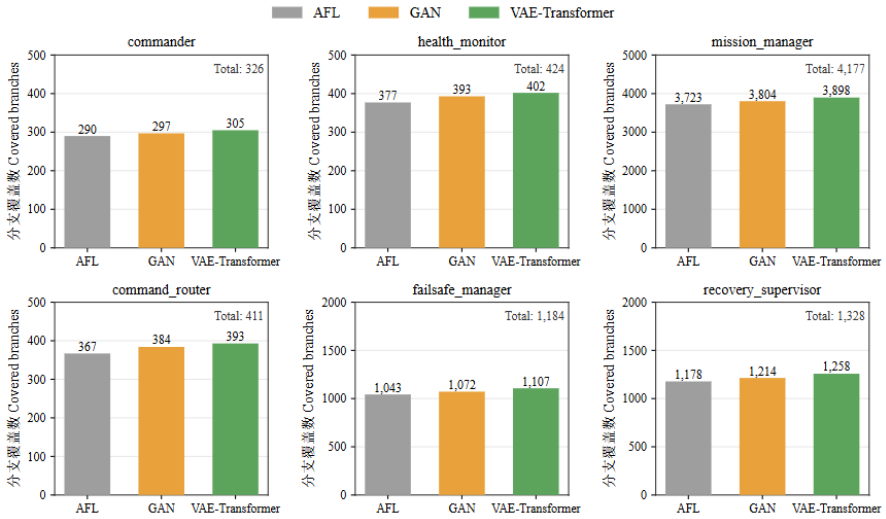

Fig. 4 Branch coverage of the compared models on key PX4 modules

English vertical-axis label: Branches covered. "Total" denotes the total number of executable branches in each program.

VAE–Transformer achieves the highest branch count on all six programs, covering 7,363 branches in total, compared with 7,164 for GAN and 6,978 for AFL. These gains of 199 and 385 branches correspond to increases of approximately 2.78% and 5.52%, respectively. On the largest program, `mission_manager`, VAE–Transformer covers 3,898 branches, exceeding GAN and AFL by 94 and 175 branches. On `recovery_supervisor`, the corresponding gains are 44 and 80 branches. Because the total number of executable branches differs among programs, absolute bar heights should not be compared directly across targets. Within every target, however, VAE–Transformer consistently outperforms both baselines. Its advantage is therefore not attributable to a single large program but is evident across programs with different sizes and functions.

Table 3 shows that AFL and GAN have similar line-coverage performance, with average coverage rates of 88.90% and 89.17%, respectively; GAN does not consistently outperform AFL on every program. In contrast, VAE–Transformer achieves the highest line coverage on all six programs, with an average rate of 94.14%, exceeding GAN and AFL by approximately 4.97 and 5.24 percentage points. In absolute terms, VAE–Transformer covers 13–32 additional lines per program compared with GAN, for a total gain of 151 lines. Compared with AFL, it covers 15–34 additional lines per program,

Table 3 Line coverage of the compared models on key PX4 modules.

Target Program	AFL	GAN	VAE–Transformer
`commander`	263/296	265/296	278/296
`health_monitor`	421/471	419/471	444/471
`mission_manager`	580/652	585/652	614/652
`command_router`	404/457	406/457	430/457
`failsafe_manager`	501/563	499/563	531/563
`recovery_supervisor`	526/592	529/592	557/592

totaling 159 lines. The largest gain over GAN occurs on `failsafe_manager`, where line coverage increases by approximately 5.68 percentage points. Even on `commander`, which shows a smaller gain, the increase remains approximately 4.39 percentage points. These results suggest that VAE–Transformer consistently explores more code regions across programs of different sizes and functions, supporting its effectiveness in improving test-case diversity and program-state-space exploration. Taken together, the two figures and the table show consistent advantages in edges covered, branches covered, and line-coverage rates. The edge-coverage curves indicate that the method continues to generate inputs that trigger different execution paths. The branch results show that these inputs activate more control-flow branches, while the line results demonstrate access to additional code regions. Across modules and coverage metrics, VAE–Transformer therefore achieves more extensive code exploration with greater efficiency, improving the effectiveness of fuzzing for complex flight-control software.

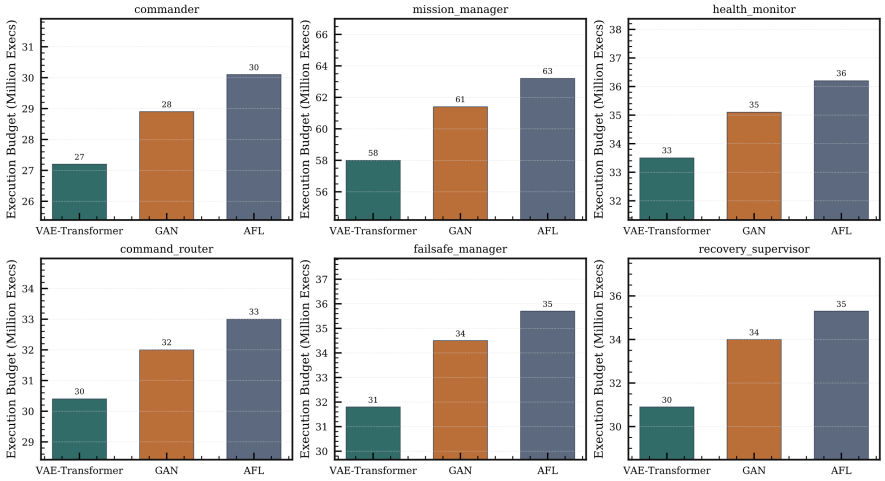

Fig. 5 Execution-budget comparison across six target programs

As shown in Fig. 5, the execution budgets of VAE–Transformer on the six targets are 27.2, 58.0, 33.5, 30.4, 31.8, and 30.9 million executions, respectively, all lower

than those of GAN and AFL. Relative to AFL, the reductions are approximately 9.6%, 8.2%, 7.5%, 7.9%, 10.9%, and 12.5%, respectively. These results suggest that the performance gains arise from more effective generated seeds rather than greater execution expenditure. Figures 3–5 and the supplementary line-coverage statistics indicate consistent advantages in coverage breadth, coverage depth, and execution efficiency. Across the six targets, the proposed model covers an average of 26.5 additional lines, 72.5 additional edges, and 64.2 additional branches, while reducing the average execution budget by approximately 9.4%. The method thus improves fuzzing not only on a single program but also across a range of airborne software targets, indicating favorable generalization and consistency.

5.4 Ablation Study

To assess whether VAE distribution modeling, Transformer contextual modeling, and gated fusion are jointly necessary, ablation experiments were conducted along two dimensions: architectural components and fusion strategies. Tables 4 and 5 report the quantitative results for the different configurations and examine the contribution of each design element to the overall framework.The ablation experiments were conducted on commander. Tables 4 and 5 were obtained from two independent batches of runs, each including its own full-model configuration as the reference; the "difference from best" columns are computed against the reference within the same batch. Because fuzzing is stochastic, the absolute values of the full model differ slightly from those in Table 2.

Table 4 Ablation of architectural components.

Model	EC	LC	BC	Execution budget (M)	Edge difference from best	Branch difference from best
VAE	228	274	302	28.9	-7	-9
Transformer	226	271	298	29.6	-9	-13
VAE–Transformer	235	282	311	26.5	0	0

Table 5 Ablation of fusion strategies.

Fusion strategy	EC	LC	BC	Execution budget (M)	Edge difference from best	Branch difference from best
No fusion	225	270	297	29.9	-10	-14
Addition	229	275	302	28.6	-6	-9
Concatenation	231	277	305	27.9	-4	-6
Gated fusion	235	282	311	26.5	0	0

EC: edges covered; LC: lines covered; BC: branches covered; M: million executions.

Tables 4 and 5 present the component and fusion-strategy ablations. In the component ablation, VAE–Transformer achieves the best performance on all three coverage metrics, covering 235 edges, 282 lines, and 311 branches. It also requires the smallest execution budget, at 26.8 million executions. Compared with VAE and Transformer, its branch coverage increases by 2.32% and 3.69%, respectively, while its edge coverage increases by 2.63% and 3.54%. These results suggest that combining VAE representation learning with Transformer sequence modeling enables the generation of more effective test inputs, discovering more paths with fewer executions. The architectural design therefore contributes materially to fuzzing coverage. The fusion-strategy ablation in Table 5 further shows that adaptive gated fusion provides the best overall performance, covering 235 edges, 282 lines, and 311 branches. It outperforms no fusion, static addition, and concatenation, while requiring the smallest budget of 26.5 million executions. Compared with no fusion, gated fusion improves edge, line, and branch coverage by 4.44%, 4.44%, and 4.71%, respectively. Its coverage advantage also persists over static addition and concatenation. These findings suggest that gating more effectively coordinates complementary information from the different representations and suppresses redundant or uninformative features, improving test-input generation and path-exploration efficiency. Overall, the performance improvement arises from the combined effects of the model architecture and fusion mechanism, rather than a single component. The ablation results support the effectiveness of the proposed design in both coverage and execution efficiency.

6 Conclusions

This paper proposed a VAE–Transformer-based method for automated airborne software test case generation to address inadequate input-structure preservation, insufficient state-semantic modeling, and limited coverage efficiency. The work has two principal aspects. First, semantic-extraction-based VAE latent modeling reorganizes binary inputs into an event–field matrix and explicitly encodes key flight-control state dimensions in an operational-state semantic conditioning vector. A conditional VAE learns the latent distribution of high-value test cases under these constraints, separating state-semantic conditioning from sample diversity. Second, event-aware gated Transformer generation adaptively coordinates conditioning semantics and stochastic latent information at each event position. A Transformer encoder models inter-event dependencies and generates complete test cases in parallel, with all components jointly optimized end to end under a unified objective. Experiments show that the proposed method improves airborne software fuzzing. On `commander`, edge, line, and branch coverage improve by approximately 4.9, 5.1, and 4.6 percentage points, respectively, over AFL, while the execution budget required to reach the target coverage decreases by approximately 9.6%. Comparisons across six representative targets further show consistent advantages in coverage breadth, coverage depth, and execution efficiency. Ablation experiments confirm the contributions of VAE distribution modeling, Transformer contextual modeling, and gated fusion to overall performance. The method has several limitations. The experiments primarily concern fixed-length structured inputs with clear field boundaries. Its applicability to variable-length protocols, deeply

nested inputs, and more complex interaction constraints requires further investigation. Future work will expand the training corpus and target-program diversity and explore a unified representation for variable-length structured inputs, with the aim of improving test-case effectiveness, program-state-space exploration, and the ability to trigger latent defects.

Statements and Declarations

Funding

This work was supported by the XX

Conflict of interest:

The authors declare that they have no competing interests.

Data availability

The datasets generated and analysed during the current study are available from the corresponding author on reasonable request.

Code availability

The code used for model training, test case generation, and experimental evaluation is available from the corresponding author on reasonable request.

Competing interests

The authors declare that they have no competing interests.

Author contributions

All authors read and approved the final manuscript.

References

[1] Watkins CB, Walter R. Transitioning from federated avionics architectures to integrated modular avionics. In: 2007 IEEE/AIAA 26th Digital Avionics Systems Conference (DASC). IEEE; 2007. p. 2.A.1–1–2.A.1–10.

[2] RTCA. Software Considerations in Airborne Systems and Equipment Certification. Washington, DC: RTCA, Inc.; 2011. DO-178C.

[3] SAE International. Guidelines for Development of Civil Aircraft and Systems. Warrendale, PA: SAE International; 2010. ARP4754A.

[4] Hayhurst KJ, Veerhusen DS, Chilenski JJ, et al. A practical tutorial on modified condition/decision coverage. Hampton, VA: NASA Langley Research Center; 2001. NASA/TM-2001-210876.

[5] Manès VJM, Han H, Han C, et al. The art, science, and engineering of fuzzing: a survey. IEEE Transactions on Software Engineering. 2021;47(11):2312–2331.

[6] Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2016. p. 1032–1043.

[7] Kim H, Ozmen MO, Bianchi A, et al. PGFUZZ: Policy-guided fuzzing for robotic vehicles. In: Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS). Internet Society; 2021. .

[8] Wang Y, Jia P, Liu L, et al. A systematic review of fuzzing based on machine learning techniques. PLoS ONE. 2020;15(8):e0237749.

[9] Hu Z, Shi J, Huang Y, et al. GANFuzz: A GAN-based industrial network protocol fuzzing framework. In: Proceedings of the 15th ACM International Conference on Computing Frontiers. ACM; 2018. p. 138–145.

[10] Godefroid P, Peleg H, Singh R. Learn&Fuzz: Machine learning for input fuzzing. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society; 2017. p. 50–59.

[11] Wang Y, Wu Z, Wei Q, et al. NeuFuzz: Efficient fuzzing with deep neural network. IEEE Access. 2019;7:36340–36352.

[12] Gan S, Zhang C, Qin X, et al. CollAFL: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy. IEEE Computer Society; 2018. p. 679–696.

[13] Lemieux C, Sen K. FairFuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM; 2018. p. 724–735.

[14] Qian R, Zhang Q, Fang C, et al. DiPri: Distance-Based Seed Prioritization for Greybox Fuzzing. ACM Transactions on Software Engineering and Methodology. 2025;34(1):26:1–26:39. https://doi.org/10.1145/3654440.

[15] Böhme M, Pham VT, Nguyen MD, et al. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2017. p. 2329–2344.

[16] Li Y, Chen B, Chandramohan M, et al. Steelix: Program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM; 2017. p. 627–637.

[17] Aschermann C, Frassetto T, Holz T, et al. REDQUEEN: Fuzzing with input-to-state correspondence. In: Proceedings of the Network and Distributed System Security Symposium. Internet Society; 2019. .

[18] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy. IEEE Computer Society; 2018. p. 711–725.

[19] Yun I, Lee S, Xu M, et al. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium. USENIX Association; 2018. p. 745–761.

[20] Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: Fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy. IEEE Computer Society; 2018. p. 697–710.

[21] Wang J, Chen B, Wei L, et al. Superion: Grammar-aware greybox fuzzing. In: Proceedings of the 41st International Conference on Software Engineering. IEEE/ACM; 2019. p. 724–735.

[22] Dutra R, Gopinath R, Zeller A. FormatFuzzer: Effective Fuzzing of Binary File Formats. ACM Transactions on Software Engineering and Methodology. 2024;33(2):53:1–53:29. https://doi.org/10.1145/3628157.

[23] Natella R. StateAFL: Greybox Fuzzing for Stateful Network Servers. Empirical Software Engineering. 2022;27(7):191. https://doi.org/10.1007/s10664-022-10233-3.

[24] Qin S, Hu F, Ma Z, et al. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing. ACM Transactions on Software Engineering and Methodology. 2023;32(6):160:1–160:26. https://doi.org/10.1145/3580598.

[25] Bamohabbat Chafjiri S, Legg P, Hong J, Tsompanas MA. Vulnerability Detection through Machine Learning-based Fuzzing: A Systematic Review. Computers & Security. 2024;143:103903. https://doi.org/10.1016/j.cose.2024.103903.

[26] Zhao H, Li Z, Wei H, et al. SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In: 2019 IEEE 12th International Conference on Software Testing, Verification and Validation. IEEE Computer Society; 2019. p. 59–67.

[27] Liu X, Li X, Prajapati R, Wu D. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33; 2019. p. 1044–1051.

[28] Ye G, Tang Z, Tan SH, et al. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. ACM; 2021. p. 435–450.

[29] Zong P, Lv T, Wang D, et al. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: Proceedings of the 29th

USENIX Security Symposium. USENIX Association; 2020. p. 2255–2269.

[30] Rajpal M, Blum W, Singh R.: Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596. Available from: https://arxiv.org/abs/1711.04596.

[31] Chen Y, Ahmadi M, Farkhani RM, et al. MEUZZ: Smart seed scheduling for hybrid fuzzing. In: Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID). USENIX Association; 2020. p. 77–92.

[32] Wang J, Song C, Yin H. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In: Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS). Internet Society; 2021. .

[33] Huang H, Chiu HC, Shi Q, Yao P, Zhang C. Balance Seed Scheduling via Monte Carlo Planning. IEEE Transactions on Dependable and Secure Computing. 2024;21(3):1469–1483. https://doi.org/10.1109/TDSC.2023.3285293.

[34] Qin Z, Fan J, Liu X, et al. Effective fuzzing testcase generation based on variational auto-encoder generative adversarial network. Engineering Applications of Artificial Intelligence. 2025;144:110094. https://doi.org/10.1016/j.engappai.2025.110094.