A Formal Control Plane for Fuzzing Campaigns: Leveraging Attribute Grammars for Dynamic Policy Selection

Pierciro Caliandro[1,2], Matteo Ciccaglione[1,2] and Alessandro Pellegrini[1,*]

[1]Tor Vergata University of Rome, Rome, Italy
[2]National Inter-University Consortium for Telecommunications (CNIT), Rome, Italy

Abstract

Fuzzing is an effective technique for software security testing, although the orchestration of fuzzing campaigns still largely relies on rigid, imperative control logic. Traditionally, this orchestration has been based on decision-making processes (such as when to mutate inputs, migrate seeds, or change scheduling policies) that are embedded directly into the execution engine. These monolithic architectures are difficult to adapt, extend, or formally verify. In this paper, we introduce a framework that decouples the control logic from the fuzzing mechanism by elevating the orchestration problem to a linguistic domain. We propose a formal control plane powered by Attribute Grammars, where the operational state of the fuzzing infrastructure is treated as a sentence in a formal language, and control policies are derived as semantic translations of this state. From this formal declarative specification, we automatically generate the source code of the fuzzer controller. This allows researchers to define complex, adaptive strategies, including backtracking-based recovery and epsilon-greedy exploration, in a more rigorous and maintainable way. We exercise our methodology by showcasing how to effectively orchestrate a distributed fuzzing campaign using a relatively-simple grammar.

Keywords

Fuzzing, Attribute Grammars, Policy Selection

1. Introduction

Fuzzing [1] is an automated software testing technique that is widely used in the field of security. Fuzzing involves running a specific application multiple times, injecting input data which gets mutated automatically (according to various rules) so as to observe its behaviour. The goal of fuzzing is to identify input configurations that may cause *crashes* or *assertion failures*. In security analysis, the ultimate goal is to reveal potential vulnerabilities that are difficult to detect using traditional techniques such as static analysis or linting. The effectiveness of fuzzing stems from the fact that the application under test is executed multiple times with increasingly diverse inputs, which are generated, starting from the initial *seed corpus* in a semi-valid manner (for example, by taking valid inputs and applying changes that break the syntax of the semanthics) or completely at random.

Input generation in modern fuzzers for the majority of the cases is guided by some sort of feedback information measured by stimulating the program under test. The predominant feedback information used [1, 2] is based on *coverage*: repeated executions use inputs deliberately generated to explore as many code paths as possible. Such input generation may happen by stacking random mutations (such as bit flipping, byte flipping, byte swapping and so on) on the same input or relying on the use of *grammars* [3, 4, 5] to generate inputs from the formal definition of the expected input format. Alternatively, fuzzing can be combined with *symbolic execution* techniques [6, 7] or *concolic execution* [8, 9, 10, 11], in order to identify code paths that may be difficult to reach, such as those protected by magic numbers.

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT
*Corresponding author.
pierciro.caliandro@uniroma2.it (P. Caliandro); matteo.ciccaglione@uniroma2.it (M. Ciccaglione); a.pellegrini@ing.uniroma2.it (A. Pellegrini)
0009-0001-3549-9353 (P. Caliandro); 0009-0005-7348-2674 (M. Ciccaglione); 0000-0002-0179-9868 (A. Pellegrini)

Other strategies, such as *direct fuzzing* [12], instead aim to prioritise inputs with a higher likelihood of reaching specific portions of code (the *targets*), for example to study the behaviour of the program under test around critical sections or to reduce the fuzzing cost during the software development cycle, when it is particularly important to stress the newly-introduced portions of code. Finally, *data-flow driven* strategies [13, 14, 15] can be adopted, in which *taint analysis* is used to guide the generation of inputs independently of control flow, so as to reveal execution states that are not immediately visible when considering only the coverage of the control-flow graph.

Regardless of the adopted exploration strategy, a fuzzing campaign on a particular program under test may encounter exploration difficulties related to *stagnation*: random generation or mutation might not produce inputs that extend coverage, symbolic execution could be subject to path explosion, and grammar-based approaches might face intrinsic limitations in the formalisation of the input format. Generally speaking, it is possible that the fuzzing strategy may not be able to grow the program under test's exploration beyond a certain point. In such contexts, *dynamically changing* the fuzzing strategy can facilitate an escape from stagnation [16, 17].

Another level of complexity in modern fuzzing is due to the increasing intricacies of software and the need to thoroughly explore the coverage of an executable to identify potential vulnerabilities or defects. This has led to fuzzing scenarios that require ever-greater computing resources to reduce the time spent on fuzzing campaigns. In particular, some recent studies [18, 19, 20] have demonstrated the opportunity of using *distributed fuzzing* mechanisms, in which carefully-managed distribution of inputs among the nodes involved in the fuzzing activity can enhance the overall exploratory capacity. However, this approach can also suffer from stagnation. In particular, the work in [19] highlighted that different distributed orchestration strategies can lead to varying speeds of exploring the system under test, depending on both the activity of each individual fuzzing node and the nature of the system under test itself. Even in this context, a variation of the input strategy at runtime can provide benefits.

Therefore, it is generally advisable that a fuzzing policy be governed by a level of dynamic orchestration that allows the exploration strategy to be altered based on performance metrics related to fuzzing activity. To be effective, this approach to dynamic orchestration (often also referred to as *adaptive fuzzing*) cannot ignore the observable metrics obtained during the execution of the fuzzers. However, different instances of fuzzers typically provide access to metrics in various formats or with different levels of granularity [21, 22], making the development of an orchestrator more complex.

Indeed, even the realisation of a very simple decision engine (for example, based on finite state machines [23, 24]) requires the integration and analysis of different metrics, both in terms of format and temporal nature. Moreover, the maintainability of these software artefacts can become complex, since decision logic is often based on models instantiated directly at the controller code level. Additionally, the orchestrator may make different choices, such as varying input selection strategies, modifying mutation strategies, allocating more computing resources to fuzzing instances that demonstrate greater effectiveness, or changing the fuzzing tool used in a subset of nodes within a distributed fuzzing system. Decision logic can have arbitrary complexity, making the maintainability of the orchestrator more complicated.

In this work, we propose and explore an alternative approach to the design, realisation, and management of fuzzer orchestration strategies, based on the use of *formal languages*, specifically *Attribute Grammars* [25]. Compared to current methodologies that often rely on complex heuristics or machine learning models, we propose a paradigm shift that involves defining a *Formal Control Plane* (FCP), which utilises formal languages to express in a rigorous yet concise manner the state transitions and sequences of actions that an orchestrator must undergo to coordinate multiple fuzzer instances effectively. Furthermore, by exploiting formal languages, the acquisition of metrics produced by fuzzers can be based on parsers automatically generated from grammars that describe the format of the data that can be acquired, for example, from log files.

Defining a FCP for the dynamic (distributed) orchestration of fuzzing systems simplifies prototyping and implementation, and also paves the way for techniques of *formal verification*, allowing one to prove, with a high degree of mathematical certainty, the correctness, completeness, and optimality of the orchestration strategy even before its execution. Furthermore, the use of formal languages

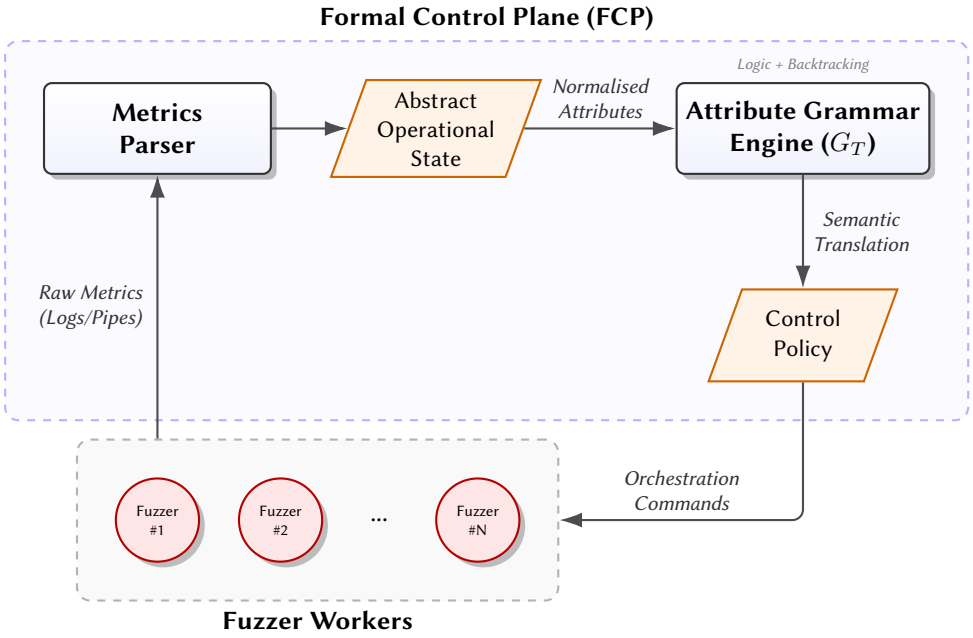

Figure 1: System Architecture

allows for greater flexibility and adaptability of the strategy in response to the dynamics of the target under analysis, while ensuring complete *traceability* and *explainability* of the decisions made by the orchestrator. Moreover, defining orchestration rules through formal languages is more concise, making it easier to explore different orchestration strategies.

Overall, the general system architecture proposed in this paper is depicted in Figure 1. We consider a group of fuzzers that can run either on a single node or in a distributed environment, which fuzz a target application according to a specific policy. The policy can be changed at runtime, either by modifying the internal parameters of the fuzzers or by starting a different fuzzing instance that executes according to a different exploration objective. Every fuzzer emits raw metrics in the form of logs, which are written to disk or output on a pipe. The FCP is a software component that is automatically generated from an Attribute Grammar specification.

The generated FCP is composed of several automatically-generated components. The first one is a parser that reads the raw metrics output by the fuzzers and normalises them into an internal notation. This notation is internally used to abstract the current operational state of the group of fuzzers, represented as a set of grammar attributes. These attributes are used by an automatically-generated Attribute Grammar Engine G_T that implements the decision logic based on the rules of the grammar. Through semantic translations, these rules convert the current abstract operational state into a target control policy, which is materialised into a set of orchestration commands sent to the group of fuzzers.

To assess the ease of implementing a FCP for complex fuzzing strategies, we demonstrate the creation of a FCP based on formal languages in a distributed fuzzing system. We utilise the so-generated FCP to guide a distributed fuzzing campaign, using a subset of benchmarks taken from the Google OSS-Fuzz suite [26].

The remainder of this paper is organised as follows. Section 2 discusses the methods and tools used to realise our proposal. The organisation and logic of the FCP are presented in Section 3. The experimental evaluation, which presents an FCP implementation in accordance with our strategy, is reported in Section 4. Finally, related Work is discussed in Section 5.

2. Background and Preliminaries

As mentioned, our definition of the FCP is based on Attribute Grammars. For the design and realisation of the FCP, we rely on the Tahr framework [27], which enables the use of an Attribute Grammar to generate code for parsing, generating, and translating text across different Attribute Grammars. Our case study is based on orchestrating a distributed fuzzing campaign. In this Section, we illustrate the methods and tools that are used as the basis of this paper. We present example code that uses the Tahr grammar syntax. We refer the reader to [27] for a description of the syntax, although it is inspired by Bison [28] and should be easily interpretable nevertheless.

2.1. Attribute Grammars and Tahr

An Attribute Grammar is formally specified as a quadruple $AG = \langle G, A, R, C \rangle$, where each element is defined as follows. The underlying context-free grammar (CFG) is $G = \langle N, T, P, S \rangle$, where N represents the set of non-terminal symbols, T represents the set of terminal symbols, P the set of production rules, and S the start symbol of the grammar. The set A denotes the collection of attributes, which, for each grammar symbol $X \in N \cup T$, is partitioned into two disjoint subsets: *inherited attributes* and *synthesised attributes*. Typically, only synthesised attributes are associated with terminal symbols, whereas non-terminal symbols may be equipped with both inherited and synthesised attributes.

The set R consists of semantic rules, each of which is associated with a grammar production $p \in P$. More formally, for any production of the form:

$$X_0 \to X_1 X_2 \dots X_n, \quad \text{with } X_i \in N \cup T, \quad 0 \leq i \leq n \tag{1}$$

semantic rules are equations defined in terms of attribute occurrences associated with symbols in the production, such as:

$$X_i.a := f(X_{j_1}.a_1, X_{j_2}.a_2, \dots, X_{j_k}.a_k) \tag{2}$$

where $X_i.a$ denotes the attribute a of the grammar symbol X_i, and f represents an attribute evaluation function.

Finally, C denotes the set of attribute constraints or conditions, typically formal predicates defined over attribute domains, which are employed to enforce semantic consistency and correctness within the model.

Attribute grammars belong to the broader class of generative grammars, as they extend the formalism of CFGs by augmenting syntactic production rules with semantic attributes and associated evaluation rules. This extension enables semantic analysis to be integrated directly into syntactic specifications. Their classification as generative formalisms arises from their explicit constructive capacity: attribute grammars generate not only syntactic structures—typically represented as abstract syntax trees (ASTs)—but also systematically computed attribute values that encode semantic information and propagate it throughout these structures.

In contrast to descriptive grammars, which characterise attested linguistic usage, and prescriptive grammars, which enforce correctness rules, attribute grammars provide a formal mechanism for defining semantics via attribute computation schemes attached to grammar productions. Through this systematic association of semantic functions with syntactic rules, they assign meanings compositionally, thus offering a precise and operational semantics for the language described.

By tightly coupling syntactic derivation with semantic evaluation in a unified formal framework, attribute grammars constitute a powerful generative mechanism that can not only derive syntactically well-formed strings, but also evaluate context-sensitive semantic properties. Typical applications include type checking, expression evaluation, code generation, and other translation and analysis tasks. Consequently, attribute grammars represent a robust and expressive formalism within the theory of generative languages and formal language semantics.

Attribute Grammars are used by the *Translational Attribute Handler and Rewriter* (Tahr) framework [27], which takes Attribute-Grammar specifications and can automatically generate a set of software artefacts capable of parsing, generating, and translating text compliant with those grammars.

In Tahr, an Attribute Grammar is used to manipulate text across a hierarchy of three interdependent levels within the grammar specification: the *Syntactic Level*, the *Semantic Level*, and the *Translational Level*. The syntactic level defines the formal structure of expressions via production rules, while the semantic level assigns meaning through attributes and logical operations. The translational level, on the other hand, acts as a logical interface that systematically maps structures from a source grammar to a target grammar, governing the translation process. This translational level is key to our definition of a FCP.

A defining characteristic of Tahr is its embrace of ambiguity and stochasticity. Unlike traditional compiler tools that enforce determinism, Tahr explicitly supports ambiguous grammars by using *non-determinism* to empower stochastic text generation. If a rule is ambiguous during generation, the system randomly selects one of the valid options. In this work, this feature is essential for exploration purposes, particularly when two fuzzing configurations are deemed equivalent based on currently available metrics, or when insufficient observations have been gathered.

To manage the semantic correctness of the production rules, Tahr introduces a `revert` statement. This allows the user to define constraints within the attribute operations. In particular, if some rule based on some attributes fails to meet a user-defined semantic constraint (checked via code embedded in the grammar), the `revert` command aborts the current production and forces the selection of an alternative, ensuring that complex grammars can be realised without sacrificing semantic equivalence. This support will be exploited to implement concise policy selection rules based on constraints.

2.2. Distributed Fuzzing with MPI

In our case study, we design a FCP for distributed fuzzing, based on the architecture presented in [19]. In this work, a master node orchestrates the fuzzing campaign, managing the corpus of inputs and coordinating the activities of worker nodes. The worker nodes execute the actual fuzzing logic, running instances of the fuzzer (e.g., AFL++) against the target application.

MPI is utilised for high-speed, low-latency communication between nodes. This allows for rapid sharing of interesting test cases (inputs that trigger new code paths or crashes) across all instances. In particular, the proposed architecture explores three different input dissemination policies, which determine how the fuzzing workload and corpus are distributed among the nodes.

The first strategy is called *Selective Dissemination*. It employs a deterministic, hash-based input forwarding mechanism, such that when a fuzzer discovers an interesting input, the receiver of the input is determined based on a hash of the data and a modulo operation. This approach ensures a uniform distribution of workload across the cluster and avoids the overhead of broadcasting inputs to all nodes, although it risks sending data to busy nodes that cannot immediately exploit it.

The second strategy is called *Dynamic Dissemination with Utilisation*. Its goal is to optimise resource usage by directing new inputs to the nodes that could benefit from them the most. The utilisation is based on how much the inputs received from peers allow exploring new code paths in the system under test.

Third policy, named *Hierarchical Dissemination Across Fuzzer Clusters*, structures the fuzzing network into distinct, interconnected groups (clusters) to balance local intensity with global coverage. It prioritises tighter and more frequent synchronisation within a specific group of fuzzers while maintaining looser connections between different groups.

Finally, the paper introduces the so-called *ammuina mode*, a supplementary operational state that triggers a synchronised, all-to-all exchange of data among MPI ranks. It serves to periodically "shake up" the global state, ensuring that any valuable test cases isolated by specific policies are eventually shared with the entire cluster to break through coverage plateaus.

3. The Formal Control Plane

The foundation of the framework proposed in this work lies in the formalisation of heuristics for managing fuzzing campaigns through a control plane based on attribute grammars. In traditional

approaches to fuzzers development, the decision-making logic, e.g., determining when to change a mutation strategy or migrate (a part of) the corpus, is typically hardcoded within the execution engine. These implementations often rely on complex chains of imperative conditional instructions and ad-hoc state machines, scattered throughout the source code of the fuzzer or the fuzzer controller.

This monolithic approach presents significant critical points: it makes understanding the overall behaviour of the system difficult, it increases the risk of introducing regressions during heuristic modifications and, most importantly, risks crystallising the control policies, making it difficult to evolve or replace them without substantial refactoring of the codebase. This structural rigidity has a critical consequence: the inability to rapidly adapt exploration strategies may reduce the fuzzers' capability to identify subtle vulnerabilities or regressions, therefore limiting its usefulness as a tool for *security research and investigation*, especially when dealing with highly-complex target applications.

Another technical barrier hindering the development of advanced orchestrators is the inherent heterogeneity of the reporting mechanisms employed by various fuzzing engines. Currently, there is no *de facto* standard for exporting runtime metrics: popular tools such as AFL++ [2], LibFuzzer [29], or Honggfuzz [30] expose essential information, such as branch coverage, execution speed, or path stability, through interfaces that are profoundly different, ranging from unstructured log files and standard error outputs to memory buffers with customised data formats. As a result, the development of a control system that could unify different fuzzers requires an extensive preliminary engineering effort to realise specific adapters for each target. The data acquisition and standardisation process can be time-consuming, but it is also structurally prone to errors: small variations in the output format of a new version of a fuzzer can invalidate the (imperative) data acquisition logic, compromising the integrity or correctness of the entire control plane.

In stark contrast, our methodology abstracts the decision-making process by elevating it to a problem of *linguistic translation*. In this perspective, the instant operational state of the testing infrastructure is treated as a string belonging to a source language, which must be translated into a sequence of control actions in the target language. The adoption of code generation frameworks based on attribute grammars transforms the system into a true *algorithmic workbench*: rather than manually implementing control mechanisms, security researchers can define policy rules at a high level of abstraction.

Since it is plausible that optimal fuzzing strategies are not known beforehand but need to be studied and refined over time through empirical observation, this approach offers a substantial advantage. Indeed, the automatic generation of controller code from declarative grammatical specifications facilitates rapid iterative cycles in which orchestration logics can be modified, tested, and versioned with the same agility as compiler grammars, regardless of the complexity or scale of the underlying architecture.

In this Section, we illustrate the architectural methodologies that enable the resolution of the heterogeneity and rigidity issues discussed above through a formal approach. As an implementation reference, we adopt the Tahr framework [27], using it as a workbench based on Attribute Grammars for the automatic generation of control plans. Although the proposed approach is theoretically feasible using other tools available in the Attribute Grammars landscape (such as Silver [31], JastAdd [32] or semantic extensions of traditional parser generators like ANTLR [33]), the choice of Tahr is motivated by its specific ability to generate efficient code and its native support for translational mechanisms and backtracking, which significantly simplify the specification of resilient control policies without introducing additional overhead into the fuzzer's runtime or in the specification of the policies.

3.1. Metrics Parsing and State Normalisation

The control plane's data ingestion requires a robust mechanism for integrating the telemetry generated by fuzzing instances. Typically, fuzzing engines expose runtime metrics such as coverage bitmap density, the number of executions per second, or timestamps of the latest crashes, using semi-structured text files, pipes, or log streams. Instead of relying on ad-hoc imperative parsers, which are often error-prone and difficult to extend, our approach leverages Tahr's parser generation capabilities to formalise the syntactic structure of these data streams.

To better illustrate this process, let us consider a real operational scenario in which the fuzzing engine

Listing 1 Example Metrics Parsing Grammar

```
log_line -> KW_UPDATE PIPE coverage_field PIPE time_field
{
    /* Semantic Action: Aggregation in the global state */
    $$.new_coverage = $3.val;
    $$.timestamp    = $5.val;
}

coverage_field -> KW_COV COLON FLOAT_VAL
{
    /* Semantic Normalisation and Type Conversion:
       1. The token FLOAT_VAL is already converted into a double by the lexer.
       2. The action normalises the scale: from percentage (0-100) to scale (0.0-1.0). */
    $$.val = $3 / 100.0;
}
```

periodically emits status lines in the format `UPDATE | cov: <val> | time: <val>`. According to the proposed approach, we could construct an attribute grammar that operates on two distinct levels of normalisation, as illustrated in the example in Listing 1. The first level of normalisation is *syntactic abstraction*: grammatical rules absorb lexical variants of the format—e.g., "cov:", exposing only the abstract concept of the coverage field to the system. The second level is *semantic normalisation of types*: the actions associated with the rules transform raw representations (e.g., an integer percentage) into canonical data types for validation (e.g., a scalar in floating-point normalised between 0 and 1).

Within the architectural framework proposed in this work, the automatically-generated parser is therefore used to handle the complexity of tokenisation and type conversion. If a future version of the fuzzer were to modify the log format (for example, changing the delimiter or adding new informational fields), the adjustment would require only a local modification to the grammar rules, leaving the downstream logic that consumes the attributes unchanged.

This preliminary layer of abstraction ensures that subsequent decision-making logic operates on a clean and semantically-consistent system state, effectively decoupling the data serialisation format from the high-level control logic. This makes the *data ingestion* phase intrinsically modular and agnostic, removing technical barriers to orchestrating heterogeneous tools.

A fundamental operational consequence of this part of our architecture is the simplification of integrating new fuzzing engines. Indeed, extending the formal control plane to support some different tools does not require altering the control code (which is automatically generated): it requires only declaratively defining a dedicated grammar for its specific output format.

3.2. From Operational States to Control Actions

The fundamental principle on which our architecture is based consists of transforming the traditional feedback cycle of fuzzing into a structured process of *linguistic interpretation*. Instead of considering runtime metrics as simple variables used rigidly to implement conditional control logic, the control plane considers these metrics as components of a "sentence" that describes the current state of the infrastructure.

This sentence is therefore *analysed syntactically* and *evaluated semantically* to derive, in a deterministic but flexible manner, the most appropriate control actions to undertake. This approach differs from more traditional orchestrators that passively monitor a set of heterogeneous metrics, responding to events through static conditional patterns (i.e., "if-this-then-that") that make the writing and maintenance of control rules an excessively complex task and prone to errors.

Indeed, we supersede this traditional reactive paradigm with our proposal: we define a translational grammar G_T in which the non-terminal symbols represent abstract operational states of the system, rather than simple syntactic constructs. The parsing process, accepting as input the normalised current fuzzing state, constructs a derivation tree that dynamically maps the observed situation onto the optimal sequence of control actions, effectively transforming the orchestration problem into a *formal translation*

Listing 2 Example of State → Action Rule.

```
evaluate_state -> check_performance
{
    if ($$.utility_score < LOW_UTILITY_THRESHOLD && $$.delta_cov == 0) {
        /* Stagnation situation (no increase in coverage, low utility).
           ACTION: Activate 'Aggressive Import'.
           In this mode, the node ignores quality filters and imports
           massively input from neighbours to introduce external genetic variability,
           potentially breaking the local optimum. */
        $$.migration_policy = POLICY_IMPORT_AGGRESSIVE;
    } else {
        /* Normal situation: maintain the standard balanced exchange. */
        $$.migration_policy = POLICY_EXCHANGE_BALANCED;
    }
}
```

task from state to action.

The expressiveness of this approach is enhanced by the modelling of attributes, which act as a *bidirectional interface* between the execution environment and the grammatical logic. The attributes are classified into two distinct categories based on the direction of information flow within the derivation tree: *inherited attributes* and *synthesised attributes*. Inherited attributes constitute the input of the control plane and are populated with metrics collected from the fuzzers. Conversely, synthesised attributes represent the system's decision output.

With the completion of the parsing tree construction and attribute evaluation, the resulting values at the root determine the operational parameters for the next control iteration. These may include identifiers associated with the selection of a particular management algorithm, or for selecting any recipients for the exchange of parts of the corpus, as well as control flags to indicate the necessity of drastic reconfiguration of the workers.

To illustrate how inherited attributes (input) are transformed into synthesised attributes (actions) through grammatical logic, we consider the example reported in Listing 2. The rule `evaluate_state` takes as input the utility score (`utility_score`) and the differential coverage (`delta_cov`) from the environment. Based on these values, the grammar "translates" the state into a specific migration policy for the inputs to the test program (`migration_policy`).

By relying on this approach, the grammar describes the syntax of the operational state, while also allowing for the active computation of its implemented semantics, encapsulating control logic within declarative rules that are easily extendable.

In addition, the Tahr framework offers a distinctive feature that proves particularly valuable in contexts of uncertainty: the ability to handle syntactic ambiguity by stochastically selecting between multiple production rules. If the "phrase" describing the operational state (i.e., the set of observed metrics) is ambiguous (in the sense that it could be validly derived from multiple competing rules), the parser can resolve the conflict by choosing a derivation at random. This feature can be exploited strategically to implement *epsilon-greedy* exploration policies [34].

In fact, when the available metrics do not offer a strong discriminant between two operating modes, or when the data is stale (i.e., lacking "fresh metrics"), a purely deterministic choice would risk locking the system into a suboptimal local optimum. In such scenarios, controlled randomness enables the exploration of alternative operating modes, gathering new metrics that would otherwise be inaccessible, and ensures that the decision-maker maintains the capacity for dynamic adaptation even in the absence of strong environmental indications.

3.3. Constraint Enforcement via Backtracking

The definition of complex control heuristics based on multiple decision planes significantly benefits from the adoption of *backtracking mechanisms*, intended as strategies for *dynamic pruning* of the search space. Similar to what happens with the *cut* operator in logic programming (e.g., in Prolog [35]), this

Listing 3 Simplified Example of `revert`-based Logic.

```
decide_strategy -> try_maintain_policy
{
    /* Stagnation control */
    if ($$.time_since_last_cov > MAX_STAGNATION_TIME)
    {
        revert;
    }

    /* No stagnation: confirm current policy*/
    $$.next_policy_id = $$.current_policy_id;
    $$.recovery_trigger = 0;
}
| force_aggressive_mode
{
    /* This rule is executed only if the previous one was reverted. */
    $$.next_policy_id = POLICY_AGGRESSIVE;
    $$.recovery_trigger = 1;
};
```

paradigm allows decision rules to be specified by initially assuming the success of the most probable or desirable path, then "cutting" and invalidating that branch if runtime violations of constraints emerge. This approach drastically reduces the complexity of the control code: rather than pre-enumerating all possible error conditions in nested conditional structures, the system attempts a linear derivation and only backtracks to an alternative choice point if strictly necessary.

The Tahr framework reifies this theoretical pattern through the native command `revert`. Unlike standard attribute grammars, where a constraint violation leads to the irreversible failure of the entire parsing process, Tahr extends the paradigm by allowing locally-controlled backtracking. In the context of our control plan, `revert` is used to establish an implicit hierarchy of priorities among actions: the system attempts to apply the standard policy and, only if semantic predicates inhibit it, performs a rollback to fallback on recovery policies, ensuring convergence towards a valid configuration.

To illustrate this mechanism in practice, consider a grammatical rule designed to decide between maintaining the current strategy and activating more aggressive policies, as shown in Listing 3. The grammar initially attempts to apply the conservative production, which maintains the current policy. However, within the semantic block, a fragment of code acts as a guard: if the stagnation time exceeds a critical threshold, `revert` is invoked. The execution of `revert` forces the engine to abandon the current production, cancel side effects, and attempt the next alternative rule that, in this case, activates an aggressive policy. By using this approach, the example in Listing 3 illustrates how imperative flow control is transformed into a declarative selection of the production rule based on inherited attributes.

The `revert` approach offers greater flexibility than rigid finite-state machines, as adding new operating conditions, such as monitoring system resources or evaluating input complexity, simply requires inserting a new alternative rule with its corresponding semantic guard. In this way, the framework implicitly handles the combinatorial complexity of state transitions, ensuring that operational decisions are always derived from a rigorous and verifiable assessment of runtime attributes.

4. Case Study

In this Section, we present a case study to showcase the viability and effectiveness of our proposal. Our case study targets the distributed fuzzing infrastructure presented in Section 2.2. At runtime, the distributed infrastructure collects statistics from each fuzzing node in the system and dumps them to multiple log files.

We process these log files at runtime, thanks to the metrics parsing capabilities discussed in Section 3.1. Based on this abstracted operational state, we determine whether to switch to a different policy, using a set of heuristics encoded in our attribute grammar. These rules have been derived experimentally from multiple runs using different threshold values, highlighting the ease of prototyping of our approach.

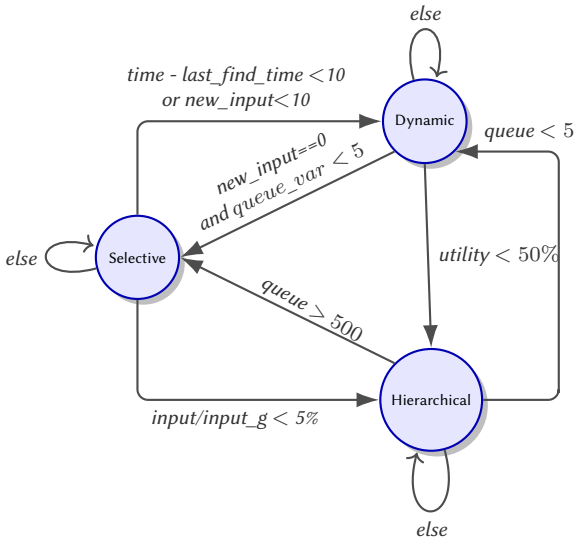

Figure 2: Finite State Automata for policy management

The generated orchestrator must choose the best policy among *Selective dissemination*, *Dynamic dissemination*, and *Hierarchical dissemination*, behaving accordingly to the finite-state machine in Figure 2, where each state corresponds to an adopted policy and the transition conditions are given by the heuristics. According to the work in [19], the following metrics can be gathered from data logs for each policy:

- *time*: time elapsed since the start of fuzzing;
- *input*: total inputs gathered during fuzzing synchronization;
- *input_good*: useful inputs gathered during fuzzing synchronisation (assumed to be a subset of the total inputs);
- *edges_total*: total number of edges between basic blocks in the application under fuzzing;
- *edges*: total number of edges covered so far;
- *time_last_find*: time at which a new edge was last discovered;
- *queue*: number of inputs currently in the queue (waiting to be processed);
- *utility*: number of fuzzers from which we receive inputs (only for the *Dynamic* policy).

Based on these metrics, we have encoded a distinct heuristic for each possible policy transition in our grammar, grounded in the policy definitions and their respective operational constraints. These heuristics are parameterised by thresholds, which we have empirically determined through multiple executions of the distributed fuzzer. By systematically parsing the generated logs to collect performance metrics, we identify the conditions and timing for switching policies to maximise exploration.

Specifically, the orchestrator switches from the *Selective* policy to the *Hierarchical* policy when the ratio of useful input to total input drops below a threshold of 5%, indicating diminished returns from hash-based dissemination and prompting a shift to cluster-based communication. A transition from *Selective* to *Dynamic* occurs if no new basic block edges are discovered for more than 10 minutes or if fewer than 10 inputs were exchanged during the last synchronisation, indicating stagnation and motivating a more dynamic dissemination strategy. When operating under the *Dynamic* policy, the system switches to *Hierarchical* if the utility metric, defined as the proportion of fuzzers contributing inputs, falls below 50%, indicating that the interactions are effectively limited to fewer than half of the total fuzzers. Conversely, a switch from the *Dynamic* to the *Selective* policy is initiated if a fuzzer's queue becomes excessively large and requires a more relaxed type of input dissemination to reduce its backlog, specifically when the number of new inputs is zero, and the variation in queue size is less than 5. Conversely, under the *Hierarchical* policy, a switch back to *Selective* is triggered if any node's input queue size exceeds 500, reflecting potential bottlenecks or overload in cluster communication. Finally, a

transition from *Hierarchical* to *Dynamic* is initiated when the queue size falls below 5, suggesting that limited inputs are processed within clusters, thus reducing the benefit of cluster-based exchanges.

4.1. Reference Implementation of the FCP

The FCP used in our case study is based on two main software components: the first one is automatically generated from a grammar specification, while the second relies on the generated code to initiate the policy selection activity and communicate the selected policy to the distributed fuzzers.

The grammar-generated component operates at the textual level and consists of a parser for the log files. Tahr automatically generates this component from the grammar specification detailed in Listings 4 and 5. Specifically, Listing 5 presents the attribute and symbol definitions. All gathered metrics are represented as a single attribute named `metric` of type `long`. To distinguish between different metrics, we assign unique attribute names to each occurrence when defining the symbols in the grammar.

Due to Tahr's requirement that each terminal symbol correspond to exactly one non-terminal symbol, we define two distinct symbols for each policy. Since we do not need to generate any output strings containing metrics, we introduce a separate terminal symbol representing the policy to switch to, which is linked to a non-terminal symbol distinct from the one used during parsing. The translation between these symbols is handled by Tahr's translational rules. Additionally, we define reduction rules that consolidate multiple occurrences of the same symbol into a single instance, reflecting the expectation that the observed policy remains consistent during parsing, since policy changes are controlled solely by us.

In Listing 4, we present all the translational, reduction, and generation rules. Each symbol has associated generation rules. All symbols include generation rules producing each of the other symbols, meaning that a policy switch is possible from any state to any other state.

Tahr's `revert` is also leveraged to prune the generation tree: its inherent non-determinism allows random selection of generation rules, and after evaluating the relevant heuristic, if a rule is deemed inapplicable, the system reverts and tries an alternative.

From this grammar, we generate the parsing and decision-making modules, which are then coupled to a simple module that scans a designated directory every five minutes to read log files produced by all fuzzers. The Tahr-generated parser produces a single symbolic representation that is then translated into an expandable symbol. Thar's grow algorithm is used to always prioritise expansions into non-terminal symbols first, so as to explore all possible policies. The outcome is a "vote" from the corresponding fuzzer. The policy switch is applied only if a majority vote favours another policy, resulting in a voting-based policy switching system.

Upon reaching a decision, we broadcast the policy change to all fuzzers. Each fuzzer periodically probes change-policy commands, ensuring timely detection without blocking fuzzing operations. Upon receipt, all fuzzers synchronously transition to the new policy, maintaining coordinated execution across the distributed system.

4.2. Experimental Results

We have assessed the effectiveness of our FCP by comparing the fuzzing behaviour of the policies presented in [19] against a dynamically-changed policy based on the heuristics encoded in the attribute grammar. We also compare against the baseline AFL++ operations. According to the architectural organisation in [19], a single fuzzing campaign employs different fuzzing instances operating according to various instrumentation methods and mutation strategies, such as AddressSanitizer (ASAN), LAF instrumentation, comparison logging (cmplog), or classical mutation strategies (referred to as "other" in the plots).

The target applications are taken from the *fuzzer-test-suite* repository[1], as they are also used in the reference Google OSS fuzz project [26]. Experiments have been conducted in a distributed environment using three identical servers equipped with Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz, 128GB RAM,

[1] https://github.com/google/fuzzer-test-suite

Listing 4 Tahr Grammar Orchestration Rules

```
%%
hierarchical_t -> HIERAR_MET ;
selective_t -> SELECTIVE_MET ;
dynamic_t -> DYNAMIC_MET ;
selective -> dynamic {
        if (!(($$.edges < EDGES_THRESH && ($$.time - $$.tlf > TIME_THRESH)) ||
        ↪  no_new_input($$.input))){
                revert;
        }
} | hierarchical {
        if (!(inputs_are_trash($$.input,$$.ig))){
                revert;
        }
} | SELECTIVE_COM ;
dynamic -> hierarchical {
        if (!($$.util < n_fuzzer/2 -1)){
                revert;
        }
} | selective{
    if (!($$.input-last_num_input == 0 && abs ($$.queue-last_queue)<QUEUE_THRESH){
            revert;
    }
} DYNAMIC_COM ;
hierarchical -> selective {
        if (!($$.queue > QUEUE_THRESH)){
                revert;
        }
} | dynamic {
        if (!($$.queue < QUEUE_LOW_THRESH)){
                revert;
        }
} | HIERAR_COM ;
%%
hierarchical_t <-> hierarchical{
        $1.queue = $$.queue;
};
selective_t <-> selective{
        $1.time = $$.time;
        $1.input = $$.input;
        $1.ig = $$.ig;
        $1.edges = $$.edges;
        $1.tlf = $$.tlf;
};
dynamic_t <-> dynamic{
        $1.util = $$.util;
};
selective_t <- selective_t,selective_t{
        $$.input = $2.input;
        $$.ig = $2.ig;
        $$.edges = $2.edges - $1.edges;
        $$.time = $2.time;
        $$.tlf = $2.time;
};
dynamic_t <- dynamic_t,dynamic_t{
        $$.util = $2.util;
};
hierarchical_t <- hierarchical_t,hierarchical_t{
        $$.queue = $2.queue;
};
```

and running Ubuntu 22.04 as the operating system. To validate our proof of concept, we have selected the four applications reported in Table 1 as a reference for different classes of applications. In the Table, we also report the initial policy configured at system startup when the FCP is employed: we have used different initial policies to illustrate that, independently of the initial configuration, the FCP is able to adapt the campaign at runtime to different (more efficient) policies.

The results of our experiments are provided in Figures 3–6. Every 5 minutes, we have collected

Listing 5 Tahr Grammar Definitions

```
%{
        #include "utils.h"
%}

%%
%token SELECTIVE_MET,DYNAMIC_MET,HIERAR_MET,SELECTIVE_COM,DYNAMIC_COM,HIERAR_COM;

%attribute metric,long,!![+-]?[0-9]+!!;

%symbol selective_t,metric,time,metric,input,metric,ig,metric,edges,metric,et,metric,tlf,metric,queue;
%symbol dynamic_t,metric,t,metric,input,metric,ig,metric,edges,metric,et,metric,tlf,metric,util,metric,queue;
%symbol hierarchical_t,metric,time,metric,input,metric,ig,metric,edges,metric,et,metric,tlf,metric,queue;

//Parsable symbols
%symbol selective,metric,time,metric,input,metric,ig,metric,edges,metric,tlf;
%symbol dynamic,metric,util;
%symbol hierarchical,metric,queue;

%%

SELECTIVE_MET: "selective" " " time "," input "," ig "," edges "," et "," tlf "," queue ;
DYNAMIC_MET: "dynamic" " " t "," input "," ig "," edges "," et "," tlf "," util "," queue ;
HIERAR_MET: "hierarchical" " " time "," input "," ig "," edges "," et "," tlf "," queue ;


SELECTIVE_COM: "selective" ;
DYNAMIC_COM: "dynamic" ;
HIERAR_COM: "hierarchical" ;

```

Target Application	Input Type	Application Class	Starting Policy
`libpng`	image/png	PNG library	*Hierarchical*
`harfbuzz`	font/otf	Font library	*Hierarchical*
`freetype2`	font/ttf	Font library	*Selective*
`re2`	string/regexp	Regexp library	*Selective*

Table 1
Benchmark summary

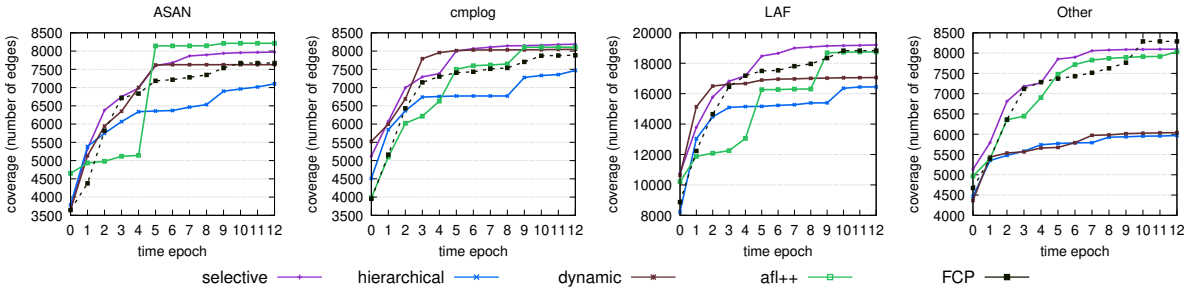

Figure 3: Results for `freetype2` benchmark

statistics associated with the number of edges collectively explored by the fuzzers—we refer to this 5-minute interval as an epoch. In the plots, we show the evolution over time of the explored edges for each considered configuration.

As can be seen, different policies yield different exploration throughput over time, also based on the nature of the target binary. Some policies clearly bump into stagnation (see, e.g., Figure 5 for the LAF and "other" cases). In most configurations, the solution that adopts our FCP is consistently able to switch at runtime to the policy that, within a certain interval, is producing a significant increase in the explored edges, thus avoiding stagnation. For example, in Figure 4 (in the "other" case), the system

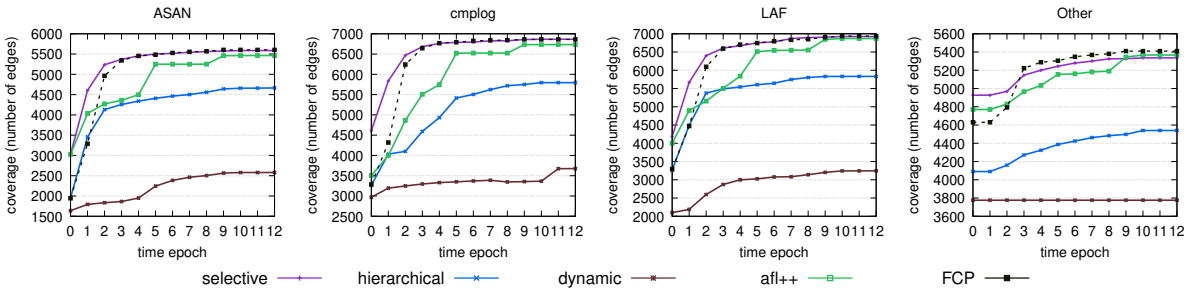

Figure 4: Results for `harfbuzz` benchmark

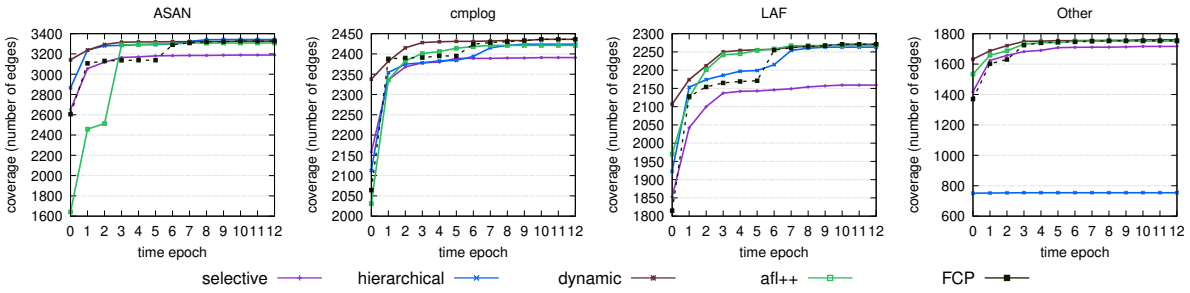

Figure 5: Results for `libpng` benchmark

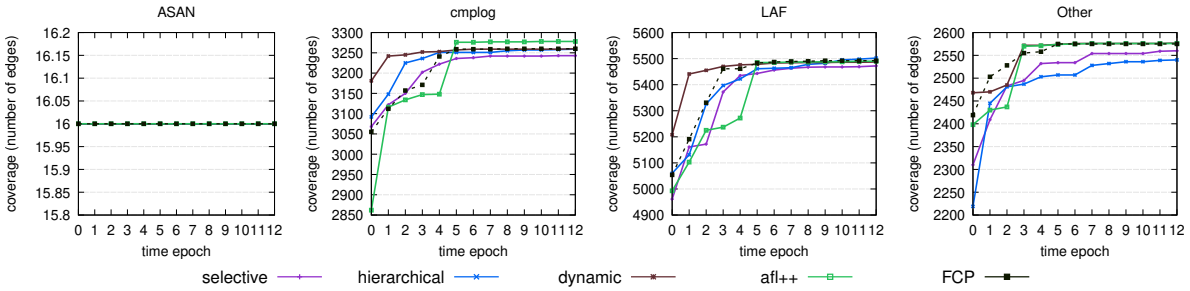

Figure 6: Results for `re2` benchmark

starts using the hierarchical strategy, which shows a reduced efficiency, but after two epochs the FCP switches to the selective strategy, which allows to catch up with the exploration and quickly deliver improved results.

In some scenarios (see, e.g., Figure 3 in the ASAN case), more classical fuzzing strategies, e.g. based on AFL++, deliver a final coverage which is higher than that of any policy. This is also reflected in the FCP case: since no policy can deliver a better result than AFL++, our approach similarly cannot. At the same time, it is interesting to note that the exploration is much faster at the beginning of the campaign. This is particularly relevant if fuzzing is used in time-constrained environments, such as when it is employed for automated testing in CI/CD pipelines. In this case, the goal would be to explore as fast as possible a large part of the executable, possibly after every included patch. By switching to the most effective policy, it is possible to explore a non-minimal part of the system under test in a reduced amount of time.

All these results demonstrate that our grammar-encoded heuristics are capable of accurately capturing the runtime dynamics of the different policies and making the proper switch when required. In cases where a policy dominates all the others (see, e.g., Figure 4), the FCP-based controller converges to this policy in a reduced number of epochs, hence yielding faster exploration of the target applications.

It is also interesting to note that, if the stagnation is common to all considered policies, FCP does not

incur any additional overhead that could limit its exploration. This aspect can be observed in Figure 6, for the ASAN case.

Moreover, our experiments also confirm that convergence to the best policy is independent of the starting configuration. For instance, observing Figures 3 and 6, we note that the most effective policy is *Selective*. By starting either in the hierarchical or dynamic policy, there will be a time when the FCP decides to switch to the selective one and sticks there for the rest of the campaign.

5. Related Work

The literature on software testing and cybersecurity has recently explored several possibilities to enhance the efficiency of fuzzing, which are slightly related to the goals we have set for this paper. In general, we can classify these proposals into three different directions: grammar-based input generation, algorithmic adaptivity, and orchestration frameworks.

Several works [36, 37, 38, 3] have focused on integrating formal grammars into the fuzzing loop to improve the generation of structured inputs, to overcome the limitations of traditional bit-level mutation strategies. These works essentially rely on context-free grammars (CFGs) or automata to guide the fuzzer's exploration. The common approach is to model the input fed into the system under test, producing valid test cases that can pass the initial parsing stages of the target application and reach deeper program logic. This body of work demonstrates the power of linguistic formalisms in fuzzing, but the fundamental difference with our proposal lies in the object of the grammar. In particular, we rely on attribute grammars not to generate (semi) valid inputs to the target, but to operate on the meta-control level of the fuzzing campaign, using formal languages to govern the behaviour of the testing infrastructure rather than the structure of the tests.

Our proposal also shares the ultimate goals of algorithmic adaptivity. Works in this direction aim to improve upon static fuzzing by dynamically adapting the system under test's exploration, mainly through numerical optimisation or learning algorithms. In [16], the authors employ evolutionary algorithms to tune parameters based on runtime feedback. Similarly, in [17], the authors use Multi-Armed Bandit (MAB) strategies to select mutation operators. Although these systems share our goal of runtime adaptability, they rely on black-box numerical solvers or predefined heuristic plugins. Our work differs in that it elevates the control logic to a white-box linguistic level. Instead of optimising some scalar objective function, our Attribute Grammar-based control plane parses the multi-dimensional state of the fuzzing infrastructure to derive complex, semantic control actions, offering a level of expressivity and verifiability that is absent in statistical approaches.

Finally, the orchestration capabilities of our FCP compare to other modular frameworks proposed in the literature, such as [21, 39, 40]. These frameworks focus on the structural modularity of fuzzing campaigns by providing the infrastructure to plug in different fuzzers, schedule tasks, or aggregate metrics. In this sense, these works are orthogonal to our main focus, as they mainly solve the engineering challenges of running multiple fuzzers. Indeed, their orchestration logic is hard-coded into the platform's core. All these proposals could benefit from our approach, as we are most interested in logical modularity by decoupling the decision-making logic from the execution engine.

Overall, the existing literature extensively discusses the use of grammars for input generation and statistical methods for adaptive scheduling. At the same time, there is a significant lack of research formalising the orchestration logic itself, which we addressed by introducing a formal control plane that treats the operational state of a fuzzing campaign as a language to be parsed.

6. Conclusions

In this work, we have explored the viability of replacing rigid, imperative control logic with a verifiable Formal Control Plane grounded in Attribute Grammars. We have treated the operational state of a fuzzing infrastructure as a sentence to be parsed and translated, and we have demonstrated how it is possible to automatically generate a runtime controller. In this way, we have decoupled policy definition

from execution mechanics. As we have experimentally shown, this approach enables the construction of effective orchestrators for fuzzing campaigns, which can also be easily based on the fine-tuning of decision-making heuristics, thanks to the more concise specification of the Formal Control Plane.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly in order to: Grammar and spelling check. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] R. McNally, K. Yiu, D. Grove, D. Gerhardy, Fuzzing: The State of the Art, Technical Report DSTO–TN–1043, Department of Defence, Australian Government, 2012.

[2] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, AFL++ : Combining incremental steps of fuzzing research, in: Proceedings of the 14th USENIX Conference on Offensive Technologies, WOOT'20, USENIX Association, Santa Clara, CA, USA, 2020, pp. 1–12. doi:10.5555/3488877.3488887.

[3] P. Godefroid, A. Kiezun, M. Y. Levin, Grammar-based whitebox fuzzing, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, ACM, New York, NY, USA, 2008, pp. 206–215. doi:10.1145/1375581.1375607.

[4] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, S. Asryan, Grammar-based fuzzing, in: 2018 Ivannikov Memorial Workshop (IVMEM), IEEE, 2018, pp. 32–35. doi:10.1109/ivmem.2018.00013.

[5] R. Hodován, A. Kiss, T. Gyimóthy, Grammarinator: a grammar-based open source fuzzer, in: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, ACM, New York, NY, USA, 2018, pp. 45–48. doi:10.1145/3278186.3278193.

[6] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, J. Sun, SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18-W, ACM, New York, NY, USA, 2018, pp. 61–64. doi:10.1145/3183440.3183494.

[7] B. S. Pak, Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution, Master's thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2012.

[8] M. Mouzarani, B. Sadeghiyan, M. Zolfaghari, Detecting injection vulnerabilities in executable codes with concolic execution, in: 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS), IEEE, 2017, pp. 50–57. doi:10.1109/icsess.2017.8342862.

[9] S. Jeon, J. Moon, Dr.PathFinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search, Neural Computing and Applications 34 (2022) 10731–10750. doi:10.1007/s00521-022-07008-8.

[10] H. Tu, S. Lee, Y. Li, P. Chen, L. Jiang, M. Böhme, Cottontail: Large language model-driven concolic execution for highly structured test input generation, 2025. doi:10.48550/ARXIV.2504.17542. arXiv:2504.17542.

[11] Z. Sun, X. Li, F. He, D. Yu, Y. Zhang, Multifuzz: a collaborative fuzzing framework based on concolic execution, in: 2025 11th IEEE International Conference on Privacy Computing and Data Security (PCDS), IEEE, 2025, pp. 01–08. doi:10.1109/PCDS65695.2025.00059.

[12] M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury, Directed greybox fuzzing, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, ACM, New York, NY, USA, 2017, pp. 2329–2344. doi:10.1145/3133956.3134020.

[13] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, J. Sun, PATA: Fuzzing with path aware taint analysis, in: 2022 IEEE Symposium on Security and Privacy (SP), IEEE, 2022, pp. 1–17. doi:10.1109/sp46214.2022.9833594.

[14] A. Herrera, M. Payer, A. L. Hosking, DatAFLow : Toward a data-flow-guided fuzzer, ACM transactions on software engineering and methodology 32 (2023) 1–31. doi:10.1145/3587156.
[15] S. Bekrar, C. Bekrar, R. Groz, L. Mounier, A taint based approach for smart fuzzing, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, 2012, pp. 818–825. doi:10.1109/icst.2012.182.
[16] A. Zhou, H. Huang, C. Zhang, KRAKEN: Program-adaptive parallel fuzzing, Proceedings of the ACM on Software Engineering 2 (2025) 274–296. doi:10.1145/3728882.
[17] D. Yu, Y. Wang, C. Zhang, Y. Lan, Z. Jiang, S. Gan, Z. Ma, W. Tan, XFUZZ: A flexible framework for fine-grained, runtime-adaptive fuzzing strategy composition, Proceedings of the ACM on Software Engineering 2 (2025) 69–91. doi:10.1145/3728873.
[18] Y. Wang, Y. Zhang, C. Pang, P. Li, N. Triandopoulos, J. Xu, Facilitating parallel fuzzing with mutually-exclusive task distribution, in: J. Garcia-Alfaro, S. Li, R. Poovendran, H. Debar, M. Yung (Eds.), Security and Privacy in Communication Networks, LNICST, Springer International Publishing, Cham, Switzerland, 2021, pp. 185–206. doi:10.1007/978-3-030-90022-9_10.
[19] P. Caliandro, M. Ciccaglione, A. Pellegrini, When high-performance computing meets software testing: Distributed fuzzing using MPI, arXiv [cs.SE] (2025). doi:10.48550/arXiv.2512.01617. arXiv:2512.01617.
[20] X. Meng, D. Liu, X. Zhou, P. Lin, C. Liu, L. Zhou, W. Xie, B. Wang, P. Wang, SimFuzz: Conflict-aware parallel fuzzing via incremental path similarity clustering, in: 2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2025, pp. 539–549. doi:10.1109/issre66568.2025.00026.
[21] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, T. Wang, UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers, in: 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 2777–2794.
[22] A. Touqir, F. Iradat, W. Iqbal, A. Rakib, N. Taskin, H. Jadidbonab, O. Haas, Systematic exploration of fuzzing in IoT: techniques, vulnerabilities, and open challenges, The journal of supercomputing 81 (2025) 877. doi:10.1007/s11227-025-07371-y.
[23] Z. Pan, L. Zhang, Z. Hu, Y. Li, Y. Chen, SATFuzz: A stateful network protocol fuzzing framework from a novel perspective, Applied sciences (Basel, Switzerland) 12 (2022) 7459. doi:10.3390/app12157459.
[24] L. Yu, S. Yanlong, Z. Ying, Stateful protocol fuzzing with statemap-based reverse state selection, arXiv [cs.CR] (2024). doi:10.48550/arXiv.2408.06844. arXiv:2408.06844.
[25] D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (1968) 127–145. doi:10.1007/bf01692511.
[26] K. Serebryany, OSS-fuzz: Google's continuous fuzzing service for open source software, in: Proceedings of the 26th Usenix Security Symposium, Usenix Security'17, USENIX Association, Santa Clara, CA, USA, 2017.
[27] M. Ciccaglione, P. Caliandro, A. Pellegrini, Tahr: The generative attribute grammar framework, arXiv [cs.FL] (2025). doi:10.48550/arXiv.2512.01872. arXiv:2512.01872.
[28] C. Donnelly, R. Stallman, Bison: The Yacc-compatible parser generator, Samurai Media Ltd, Guelph, ON, USA, 2015.
[29] LLVM, libFuzzer – a library for coverage-guided fuzz testing, 2017.
[30] R. Swiecki, honggfuzz: Security oriented software fuzzer, 2023.
[31] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: An extensible attribute grammar system, Science of computer programming 75 (2010) 39–54. doi:10.1016/j.scico.2009.07.004.
[32] G. Hedin, E. Magnusson, JastAdd—an aspect-oriented compiler construction system, Science of computer programming 47 (2003) 37–58. doi:10.1016/s0167-6423(02)00109-0.
[33] T. J. Parr, R. W. Quong, ANTLR: A predicated-$LL(k)$ parser generator, Software: practice & experience 25 (1995) 789–810. doi:10.1002/spe.4380250705.
[34] R. S. Sutton, A. G. Barto, Reinforcement Learning I: Introduction, MIT Press, 1998.
[35] L. Sterling, E. Y. Shapiro, The art of PROLOG: Advanced programming techniques, Logic Programming, MIT Press, London, England, 1986.

[36] N. Bennett, W. Zhu, B. Simon, R. Kennedy, W. Enck, P. Traynor, K. R. B. Butler, RANsacked: A domain-informed approach for fuzzing LTE and 5G RAN-core interfaces, in: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA, 2024, pp. 2027–2041. doi:10.1145/3658644.3670320.

[37] J. A. Zamudio Amaya, Shaping test inputs in grammar-based fuzzing, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 2024, pp. 1901–1905. doi:10.1145/3650212.3685553.

[38] P. Srivastava, M. Payer, Gramatron: effective grammar-aware fuzzing, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '21, ACM, New York, NY, USA, 2021, pp. 244–256. doi:10.1145/3460319.3464814.

[39] H. Chen, Y. Li, B. Chen, Y. Xue, Y. Liu, FOT: a versatile, configurable, extensible fuzzing frameworku, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, 2018, pp. 867–870. doi:10.1145/3236024.3264593.

[40] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida, H. Bos, CollabFuzz: A framework for collaborative fuzzing, in: Proceedings of the 14th European Workshop on Systems Security, EuroSys '21, ACM, New York, NY, USA, 2021, pp. 1–7. doi:10.1145/3447852.3458720.