Hybrid Fuzzing of Infrastructure as Code Programs (Short paper)

Emilio Coppa
ecoppa@luiss.it
LUISS University
Italy

Daniel Sokolowski
science@d.sokolowski.org
Independent Researcher
Germany

Guido Salvaneschi
guido.salvaneschi@unisg.ch
University of St. Gallen
Switzerland

Abstract

Infrastructure as Code (IaC) has become a cornerstone of modern cloud and system deployment, enabling automated and repeatable infrastructure provisioning. However, ensuring the correctness of IaC programs remains challenging due to their complexity and dynamic nature. In particular, IaC programs can exhibit different behaviors depending on the state of the resources they manage. Since these resources are deployed on external providers, accounting for their possible states is difficult, making the testing phase particularly challenging. This paper presents HIT, a novel unit-testing framework for IaC programs that effectively tests IaC code using relevant resource states. HIT combines fuzzing and concolic execution, two effective yet previously unexplored techniques for IaC code. Our experiments confirm that HIT achieves better code coverage than state-of-the-art approaches.

CCS Concepts

• **Software and its engineering → Software verification and validation**.

Keywords

Fuzzing, Infrastructure as Code, Symbolic Execution, DevOps

ACM Reference Format:
Emilio Coppa, Daniel Sokolowski, and Guido Salvaneschi. 2025. Hybrid Fuzzing of Infrastructure as Code Programs (Short paper). In *34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25), June 25–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3713081.3731721

1 Introduction

Modern infrastructure provisioning and application deployment demand high levels of automation to accommodate rapid and frequent changes in system requirements.Infrastructure as Code (IaC) [20] has become a key DevOps practice for automated, repeatable infrastructure management, enabling software engineering methods like version control [22] and testing [34] for infrastructure code.

In the IaC space, some evolution directions are clearly recognizable. First, *declarative* IaC solutions allow developers to specify the desired state of their deployment, and the IaC framework executes the necessary steps to reach it. Second, recent advancements, such as Pulumi [28], AWS CDK [1], and CDKTF [15], have introduced a shift from DSLs like JSON and YAML to adopt general-purpose languages such as Python, TypeScript, and Java. This shift enhances abstraction, tooling support, and maintainability, simplifying complex cloud infrastructure management.

Yet, while traditional configuration scripts have a simple structure, such flexibility (1) comes at the cost of tackling the complexity of a fully-fledge programming language with control-flow structures, dynamic resource creation, and dependencies on external modules, making comprehensive testing difficult. Also, (2) unlike traditional software, IaC programs interact with cloud providers, making their behavior dependent on real-world infrastructure states. Ensuring the correctness and security of modern IaC programs is challenging. Misconfigurations and inconsistent deployments can cause severe operational issues, while existing unit and integration testing methods often fail to cover complex deployment scenarios, leaving organizations vulnerable to critical misconfigurations.

Fuzzing [38] has proven highly effective in traditional software, especially when considering *graybox fuzzers*, such as AFL [37], as proved by Google's OSS-Fuzz [12], which continuously test open-source software and has discovered thousands of bugs. These methods start with a small pool of inputs and apply random mutations, generating diverse inputs to explore program states.

Similar ideas are not applicable to IaC right away: in IaC, program execution leads to an actual deployment making unit testing time and cost prohibitive. This issue has been recently tackled by ProTI, a fuzzing framework for IaC programs that automatically mocks IaC resources, generating randomized values for infrastructure components, and incorporates oracle-based validation. Yet, randomized fuzzing alone struggles with complex IaC logic. For instance, provisioning a resource may depend on specific input combinations, which are difficult to guess with random inputs.

In traditional software, fuzzing is often combined with symbolic execution [3], a powerful testing technique used to generate inputs that precisely satisfy program conditions. By systematically exploring execution paths through constraint solving, symbolic execution enhances test coverage. However, symbolic execution often fails to scale in the presence of large and complex code. Hence, popular symbolic frameworks like KLEE [5], angr [33], and SymCC [25] are frequently paired with AFL-like tools [11] to create *hybrid fuzzers*, balancing effectiveness with scalability.

Interestingly, symbolic execution has yet to be explored in the context of IaC programs. While IaC programs are highly dynamic and rely on complex runtime systems—posing potential implementation challenges for symbolic engines—they are typically small (a few hundred LOCs) and contain few loops or other sources of *path explosion*, a major obstacle for symbolic execution.

Our contributions. In this paper, we explore the challenges of testing modern IaC programs and propose a novel approach that integrates fuzzing and symbolic execution to improve the effectiveness of IaC testing. Our contributions are as follows:

This work is licensed under a Creative Commons Attribution 4.0 International License.
ISSTA Companion '25, Trondheim, Norway
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1474-0/2025/06
https://doi.org/10.1145/3713081.3731721

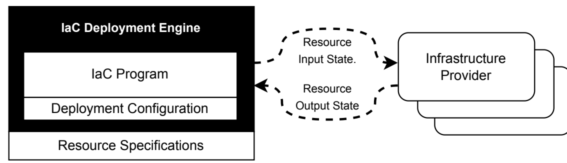

Figure 1: IaC program execution overview.

(1) We discuss the challenges of testing IaC programs, particularly code behaviors dependent on resource states, and explain why state-of-the-art (SOTA) solutions may be ineffective.
(2) We introduce HIT, a novel approach that combines random fuzzing and symbolic execution into a *hybrid fuzzing solution* tailored for testing IaC programs.
(3) We present an initial implementation of HIT, targeted at the testing of Pulumi Typescript programs. It exploits ProTI for randomized fuzzing and ExpoSE for symbolic execution.
(4) We report experimental results that shows the potential of our hybrid approach in improving the code coverage, compared to SOTA solutions, on a dataset of IaC programs.

2 Challenges in Testing of IaC Programs

This section outlines key concepts of IaC programs, showing the issues emerging with their testing.

2.1 IaC Program Execution in a Nutshell

A simplified execution workflow of an IaC program is in Figure 1. IaC programs define the *resources* required for a deployment, initializing their configuration parameters (the *resource input state*), through a *deployment configuration*, which includes essential parameters such as cloud credentials, machine types, resource limits, and security profiles. The IaC deployment engine evaluates the program and generates the API calls to request the resources. These requests are sent to infrastructure providers, such as cloud providers, to provision the necessary components. An IaC program may define a resource input state not only using static deployment configuration attributes but also the *output state* of previously deployed resources, i.e., resource allocation follows a dependency chain. The resource input and output states are defined according to a resource specification created by IaC platform developers based on the infrastructure provider documentation. This specification formally defines the attributes (and their data types) for the resource input and output states.

Figure 2 shows two (simplified) IaC programs in Typescript and Pulumi. The program on the left (*Hosting a website on AWS S3*) provisions an AWS S3 bucket configured as a static website and dynamically generates an `index.html` file with randomized content. It first creates an S3 bucket with website hosting enabled, then uses the `@pulumi/random` resource (local) provider to generate a random index for selecting a message from a given pair of words. The selected word is then used to create an `index.html` file, which is uploaded as an S3 bucket object with appropriate content type. Finally, the program exports the website endpoint URL, allowing access to the hosted HTML page. The program on the right (*Spawning two EC2 instances*) first creates a new Virtual Private Cloud (VPC) with a CIDR block of `10.0.0.0/16`. Within this VPC, it defines a subnet with a CIDR block of `10.0.1.0/24`. Then, it launches an EC2 instance (`I0`) using the specified AMI (`ami-cafecafe`) and a `t2.micro` instance type, associating it with the created subnet. Additionally, the program employs an `apply` function to monitor the state of the first instance. If `I0` reaches the `"running"` state, it triggers the creation of a second EC2 instance (`I1`) with the same AMI, instance type, and subnet configuration as `I0`, effectively creating a conditional instance deployment based on the state of the first instance.

2.2 Input dimensions for IaC programs

The behavior of an IaC program naturally depends on two main *input* dimensions:

- *Deployment configuration.* For instance, the programs in Figure 2 include several *static* deployment configuration properties, such as the bucket name `"site"` or the instance type `t2.micro`, which are used to instantiate resources during deployment. Modern IaC solutions provide abstractions, such as the `Config` object in Pulumi, to manage these configuration attributes effectively. This allows developers to switch between different values depending on the deployment context (e.g., *development* versus *production*).
- *Resource state.* Modern IaC frameworks enable developers to write programs that monitor and react to the current deployment state. For example, the program in Figure 2a generates page content based on a string determined at deployment time by the state (i.e., value) of the resource `random.RandomInt`. Similarly, the program in Figure 2b conditionally creates a second EC2 instance based on the state of the resource `aws.ec2.Instance`.

Each input dimension may require distinct testing strategies, which can be combined for greater efficacy. In this paper, we focus on testing an IaC program under different resource states. This approach is essential because, while the deployment configuration is mostly under the control of developers, resources are provided by the providers, and their internal state depends on the specific execution state of an external entity, which may be affected by numerous factors. We next examine how state-of-the-art IaC testing solutions address this issue and highlight their limitations.

2.3 Limitations of testing approaches for IaC

Testing IaC programs is challenging because simulating or emulating deployments entails a high degree of freedom in the input dimensions (Section 2.2). When focusing on the input dimension for resource state, testing of IaC programs is challenge even with relatively simple resource chains. Indeed, even when the deployment configuration is known and well-defined a developer may struggle to test all possible deployment outcomes since accurately reproducing infrastructure behavior under all scenarios is impractical – if not – unfeasible. SOTA IaC testing solutions [34] have thus explored how to exploit resource specifications to generate random yet valuable resource output states, enabling effective and efficient unit testing of IaC programs under different scenarios.

For instance, SOTA IaC testing solutions, such as ProTI [34], can effectively test the program in Figure 2a. After retrieving the specifications for `aws` resources and the (local) `random` resource from Pulumi, ProTI mocks the resources and executes the program

```
import * as aws from "@pulumi/aws";
import * as random from "@pulumi/random";

const bucket = new aws.s3.Bucket("site", {
  website: { indexDocument: "index.html" },
});
const range = { min: 0, max: 1 };
const rng = new random.RandomInt("i", range);
rng.result.apply((coin) => {
  const msg = coin % 2 == 0 ? "Hi" : "Hello";
  return new aws.s3.BucketObject("index", {
    bucket, key: "index.html",
    contentType: "text/html; charset=utf-8",
    content: msg,
  });
});
export const url = bucket.websiteEndpoint;
```

(a) Hosting a website on AWS S3.

```
import * as aws from "@pulumi/aws";
const subnet = new aws.ec2.Subnet("s", {
  vpcId: new aws.ec2.Vpc("vpc",
    { cidrBlock: "10.0.0.0/16" }).id,
  cidrBlock: "10.0.1.0/24",
});
const inst = new aws.ec2.Instance("I0", {
  ami: "ami-cafecafe", instanceType: "t2.micro",
  subnetId: subnet.id,
});
inst.instanceState.apply((state) =>
  state === "running" &&
  new aws.ec2.Instance("I1", {
    ami: inst.ami, instanceType: inst.instanceType,
    subnetId: inst.subnetId,
  })
);
```

(b) Spawning two EC2 instances.

Figure 2: Example: two simplified Pulumi programs in TypeScript.

considering various (random) values for the resource states. The values in the resource output state are generated consistently with the resource types declared in the specification, e.g., ProTI mocks the `random.RandomInt` resource and generates alternative output states for this resource. By leveraging the resource specification, ProTI learns that the output state consists of an integer and can thus test the program under different (random) integer values – hence different resource input states for `aws.s3.BucketObject`.

Yet, resource specifications often lack semantic details about how the resource input state impacts the resource output state – they typically only define the data types for each attribute of the input and output state. Also, such data types are usually not specialized, relying on primitive types, creating a semantic gap that limits SOTA IaC unit-testing solutions. For instance, the program in Figure 2b creates a second instance only when the `instanceState` of the first EC2 instance equals `"running"`. Yet, the Pulumi resource specification declares this attribute as a `string`, a primitive type, which hinders the fuzzer from generating the precise value for the program to pass the conditional and allocate the second EC2 instance. Such simple cases may be addressed through program analysis, but more complex programs require advanced techniques. Motivated by this challenge, this paper explores a unit-testing approach that leverages more sophisticated analyses to determine relevant values for fuzzing resource states.

3 Hybrid Fuzzing for IaC Programs

In this section, we present HIT (Hybrid IaC Testing), our hybrid fuzzing unit-testing framework for IaC programs written in Typescript for Pulumi.

3.1 Conceptual steps

The main conceptual phases of HIT are in Figure 3:

- **Program Loading**: The first step is to load the code of the IaC program and its dependencies, including any Pulumi package related to the resources required by the program. Moreover, the Pulumi SDK is loaded as it is necessary for the correct execution.
- **Resource Mocking**: After loading the program and its dependencies, the code is analyzed to identify resource definitions, mocking the constructors of all resource classes using Pulumi's runtime mocking. At runtime, the mocks are responsible for accepting the *resource input states* defined by the program, validating them through *resource oracles* and generating realistic *resource output states* that are consistent with the *resource specifications*.
- **Testing Loop**: The core phase of HIT is the *Testing Loop*, where the program is repeatedly tested using different resource output states. In practice, since HIT may be used for short unit-testing sessions, the testing loop can be configured to stop after a specified timeout or a set number of attempts. Across loop iterations, HIT may generate several inputs, i.e., possible alternative values to use in the output states of the resources, which are tracked by an *input queue* and consumed throughout the iterations.

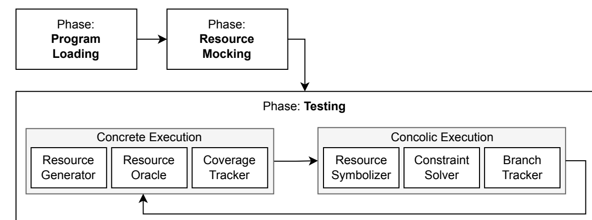

Figure 3: Main workflow of our approach.

3.2 Hybrid IaC Testing at Work

In each testing loop iteration, HIT alternates between *Concrete* and *Concolic* execution modes.

Concrete Execution. In this execution mode, HIT natively runs the program mocked during the previous phase. When the program invokes a resource builder, the related mock utilizes three main subcomponents: *resource generators*, *resource oracles*, *coverage tracker*.

For each mocked resource, HIT employs a resource generator to produce realistic resource output states. The generator evaluates the resource specification and recursively constructs a TypeScript object that complies with the expected resource type declaration. In HIT, the values used to populate the resource output state can come from two distinct sources:

- *Arbitraries*: HIT relies on arbitraries, i.e., type-specific value producers, from SOTA property-based testing frameworks when the *input* queue is empty. Arbitraries generate random values based on a random generator state, thus enabling deterministic reproduction of an execution.
- *Input Queue*: Whenever available, HIT selects a set of previously generated (but not yet tested) resource output states by the concolic execution mode. Since values generated by the concolic execution mode are intended to increase coverage, the execution path in concrete execution mode may encounter additional, previously untested resources. In such cases, HIT uses arbitraries to generate realistic output states for these new resources.

Resource Oracles are optional but can be used to validate the input states of resources, detecting inconsistencies and other problematic scenarios. Validation strategies can often be reused across different resources, and HIT provides a plugin interface that allows developers to easily define custom validation strategies. Finally, HIT tracks statement and branch coverage to assess the testing loop's effectiveness. Ideally, statement coverage increases when concolic execution successfully generates input values that lead the program down a previously unexplored branch.

Concolic Execution. HIT employs concolic execution [3], a dynamic variant of symbolic execution. Like traditional symbolic execution, it instruments the program to track computations and decision points influenced by input data (i.e., resource states in our context) and uses a constraint solver to generate alternative inputs. However, concolic execution analyzes only a single execution path, i.e., in HIT, the one explored in the last concrete execution.

In more detail, whenever the program invokes a resource constructor, HIT employs *resource symbolizers* to construct realistic objects that comply with resource specifications, similar to resource generators in concrete execution mode. Yet, unlike a resource generator, a *resource symbolizer* not only constructs the resource state but also generates a *shadow object*, where each internal attribute of the resource state is marked *symbolic*. This means that the attribute's value in a shadow object is not fixed (i.e., not yet concrete) and can initially assume any value permitted by its data type.

As the program executes computations over resource output states, HIT mirrors these operations, considering the associated symbolic shadow objects and constructing *symbolic expressions*, which represent computations on symbolic attributes. For instance, a modulo operation by 2 on a symbolic integer attribute *rng.result* would be represented by the symbolic expression $rng.result \,\%\, 2$, thus accurately modeling the computation regardless of the concrete value chosen for $rng.result$.

When the program reaches a branch, HIT first determines the alternative taken based on the current concrete input values. It then builds a symbolic expression representing the untaken direction and queries a *constraint solver* to possibly generate alternative input values that lead the program to the untaken branch. When successful, the generated input values are added to the input queue for future testing. For instance, in an untaken branch having the condition "$rng.result \,\%\, 2 == 0$", HIT queries the solver for a value satisfying the condition, obtaining, e.g., "$rng.result = 0$".

Regardless of whether the constraint solver successfully generates an alternative input, HIT continues execution along the path for the current input values, adding a symbolic expression for the taken decision to the *path constraints*. When reaching the next branch, the path constraints, in conjunction with the branch expression, are passed to the constraint solver to generate new values consistent with the path effectively explored by the current concolic execution. For instance, if the program has taken the direction associated with condition "$rng.result \,\%\, 2 \neq 0$" and then hits the untaken condition "$rng.k == 5$", it queries the solver with "$rng.result \,\%\, 2 \neq 0 \wedge rng.k == 5$".

Beyond branch decisions, HIT can also generate alternative resource output states that cause the program to consume different values from non-primitive data structures. For instance, if input values are used to access specific elements of an array, HIT employs the constraint solver to generate alternative inputs that lead to accessing different array elements or even invalid indices.

Finally, concolic execution mode tracks branch coverage to avoid generating constraint queries for already explored branches. Additionally, HIT maintains a record of all generated input values—both tested and untested—to avoid pushing duplicates in the input queue.

Running Examples. In the program in Figure 2a, HIT mocks the constructors of the resources `Bucket`, `RandomInt`, and `Bucket Object`. During the first iteration, in concrete mode, it randomly generates output states for these three resources and executes the program. Assuming it randomly selects an even integer value for $rng.result$, i.e., `coin`, it generates a `BucketObject` with the `"hi"` page content. In concolic mode, HIT reuses the same values but symbolizes all internal attributes of the resources, including the integer `coin`. Upon reaching the branch in line 10, it leverages the constraint solver to compute an alternative value for `coin` that forces execution to the opposite branch, e.g., an odd integer such as $coin = 1$. This newly generated concrete value, along with the other values used in the current execution path, is added to the input queue, so that the next testing loop iteration explores a different path. As discussed in Section 2, SOTA IaC unit testing frameworks relying solely on random fuzzing can quickly achieve the same path coverage since the branch in line 10 can easily be satisfied when considering a few alternative random values.

For the program in Figure 2b, the advantages of HIT's concolic mode over SOTA random fuzzing solutions become more evident. After mocking the constructors of the `Subnet` and `Instance` resources, HIT initially randomly fuzzes their internal values during concrete execution. This approach is unlikely to reproduce a path where a second EC2 instance is spawned. Yet, in concolic mode, HIT symbolizes the resource state attributes and, using the constraint solver, generates a value for `instanceState` equal to `"running"`. This allows HIT to explore an execution path that spawns the second EC2 instance as early as the second testing loop iteration. In contrast, SOTA solutions relying solely on random fuzzing unlikely generate the required value `"running"` for `instanceState`.

4 Implementation

The current implementation of HIT, is built by mixing and improving several key technologies: ProTI [34], a random fuzzing IaC framework; Jest, a popular JavaScript framework for efficient and effective unit testing; Fast-Check, a property-based testing framework

Pr ID	# LOC	# Res Types	Stmt Cov (%) ProTI	Stmt Cov (%) HIT	Br Cov (%) ProTI	Br Cov (%) HIT	Time (s) ProTI	Time (s) HIT
0	94	6	82.35	**100**	0	**100**	32	212
1	163	11	93.1	**100**	69.23	**100**	31	276
2	180	13	96.77	**100**	66.66	**100**	36	743
3	48	4	87.5	**100**	33.3	**100**	17	198
4	61	6	89.47	89.47	88.23	88.23	27	390
5	64	8	91.3	91.3	66.6	66.6	24	150
6	49	4	84.61	84.61	75	75	24	155
7	82	2	75	**100**	50	**100**	32	234
8	50	6	100	100	100	100	24	194
9	79	8	81.48	**96.26**	28.57	**85.71**	20	210

Table 1: Coverage and running time when running ProTI and HIT **on 10 IaC programs.**

that provides a rich set of arbitraries; Babel, for transpiling TypeScript to JavaScript; and Pulumi's runtime, which enables mocking functionalities. HIT improved the plugin interface in ProTI, allowing the integration of custom transformers, and added arbitraries to support the resource symbolizers. To support the concolic mode in HIT, we reworked several internal components of ExpoSE [19], a concolic engine based on the Jalangi2 instrumentation framework. Our improvements were necessary to adapt ExpoSE to the IaC context and integrate it with the ProTI architecture.

5 Experiments

To evaluate HIT, we generated 10 IaC programs for Pulumi TypeScript using GPT-4o, instructing it to generate code with real-world behaviors dependent on resource output states. The choice of using LLM-generated programs stems from (1) no public benchmark suite exists, (2) major LLMs are likely trained on diverse IaC code, and (3) LLMs gain popularity in IaC code generation, as explored by prior research [17, 26, 35] and encouraged by specialized IaC code generation services like Pulumi AI [27]. To improve the success rate during generation, we implemented a feedback loop that exposes syntactic errors, allowing the LLM to iteratively fix common issues. Experiments were executed on the Docker container image `node:21.1.0` with ProTI 1.3.0, using an Intel Core Ultra 7 165U CPU and 64GB RAM. Each program was tested for 107 iterations (default budget in Jest for ProTI), having HIT activate the concolic mode only in 25% of the iterations. The results of our experiments are summarized in Table 1. The programs have sizes ranging from 48 to 180 LOC and use from 2 to 13 distinct resource types. When measuring the coverage with InstabulJS, HIT is more effective than ProTI on 6 out of 10 programs. In programs #4, #5, and #9, missed conditional behaviors stem from factors beyond resource state, while in program #6, HIT is losing track of a resource state due to an imperfect instrumentation that we are trying to fix. HIT incurs significant overhead compared to ProTI (6.5× to 21×). However, HIT currently takes more than 80 seconds during the first concolic run due to inefficient ExpoSE and Jalangi2 runtime loading.

6 Related work

Software Fuzzing. Fuzzing has advanced significantly [38], with grey-box fuzzing (GBF) becoming dominant for traditional software. Unlike black-box fuzzers that mutate inputs blindly [36], GBF uses lightweight instrumentation to track coverage, prioritizing inputs exploring new paths. While GBF effectively detects bugs [24], it struggles with magic numbers, checksums [2], and structured inputs [4, 24]. Modern fuzzers address these challenges with lightweight program analysis, like approximated taint analysis [2], to bypass roadblocks or infer input grammars [4]. Consequently, HIT's random fuzzing component could be adapted into a modern GBF.

Symbolic Execution. Another well-known software testing technique is symbolic execution [3], which explores execution paths by treating inputs as symbolic variables rather than concrete values. It formulates constraints to model input-dependent program decision points and uses SMT solvers to determine values that satisfy specific conditions, possibly enabling discovery of edge cases and vulnerabilities. Examples of symbolic execution frameworks are KLEE [5], angr [33], SymCC [25], and Symbolic Path Finder [23]. While powerful, symbolic execution faces path explosion and external dependency (e.g., system calls) challenges [3]. We focus on concolic execution (a dynamic variant), leveraging IaC programs' limited size to mitigate path explosion, while resource specifications and mocking techniques help handle external dependencies.

IaC Testing. Rahman et al. [29] found limited research on IaC quality despite extensive Configuration as Code (CaC) tool studies. Guerriero et al. [13] highlighted industry's need for better IaC testing tools, while Chiari et al. [9] noted gaps in IaC static analysis research. These studies underscore the need for systematic IaC quality techniques. Pre-deployment assessment has been minimally studied: Lepiller et al. [18] introduced Häyhä for AWS CloudFormation vulnerability detection, while Cauli et al. [6, 7] used description logic for security flaw checking. However, these methods lack broad applicability, especially for provisioning tools like Pulumi and Terraform. Similarly, existing approaches for idempotency [14, 16, 31], defect prediction [8, 10], and code smell detection [21, 30, 32] have only targeted CaC. Sokolowski et al. [34] introduced ProTI, the first testing approach for IaC programs. We extend their fuzzing strategy by integrating concolic execution to improve testing effectiveness.

7 Conclusion

Ensuring IaC program correctness is challenging due to cloud environment complexity and dynamic resource state dependencies. While existing approaches provide a foundation, they fall short with intricate conditional logic and resource dependencies. To this end, we introduced HIT, a hybrid fuzzing framework combining random fuzzing with concolic execution to improve code coverage.

Acknowledgments

This work was partially supported by: project FARE (PNRR M4.C2.1.1 PRIN 2022, 202225BZJC, CUP D53D23008380006); project SETA (PNRR M4.C2.1.1 PRIN 2022 PNRR, P202233M9Z, CUP B53D23026000001). Both projects are under the Italian NRRP MUR program funded by NextGenEU. This work has also been supported by the Swiss National Science Foundation (SNSF, Grant No. 200429 and 10001777), by armasuisse Science and Technology, and by European Union's Horizon research and innovation programme (CAPE Project, Grant No. 101189899).

References

[1] Amazon Web Services. [n. d.]. Cloud Development Framework: AWS Cloud Dev.

Kit. https://aws.amazon.com/cdk/.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (2018). doi:10.1145/3182657

[4] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. https://www.usenix.org/system/files/sec19-blazytko.pdf

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[6] Claudia Cauli, Meng Li, Nir Piterman, and Oksana Tkachuk. 2021. Pre-deployment Security Assessment for Cloud Services Through Semantic Reasoning. In *33rd Int. Conf., CAV 2021 (Lecture Notes in Computer Science)*. doi:10.1007/978-3-030-81685-8_36

[7] Claudia Cauli, Magdalena Ortiz, and Nir Piterman. 2022. Actions over Core-closed Knowledge Bases. In *11th International Joint Conference, IJCAR 2022, (Lecture Notes in Computer Science)*. doi:10.1007/978-3-031-10769-6_17

[8] Wei Chen, Guoquan Wu, and Jun Wei. 2018. An Approach to Identifying Error Patterns for Infrastructure as Code. In *2018 IEEE Int. Symposium on Software Reliability Engineering Workshops*. doi:10.1109/ISSREW.2018.00-19

[9] Michele Chiari, Michele De Pascalis, and Matteo Pradella. 2022. Static Analysis of Infrastructure as Code: A Survey. In *IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022*. IEEE, 218–225. doi:10.1109/ICSA-C54293.2022.00049

[10] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian A. Tamburri. 2022. Within-project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Trans. Software Eng.* 48, 6 (2022), 2086–2104. doi:10.1109/TSE.2021.3051492

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[12] Google. 2019. Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz.

[13] Michele Guerriero, Martin Garriga, Damian Andrew Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*. doi:10.1109/ICSME.2019.00092

[14] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting Reliable Convergence for Configuration Management Scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*. doi:10.1145/2983990.2984000

[15] HashiCorp. [n. d.]. CDK for Terraform. https://developer.hashicorp.com/terraform/cdktf. Accessed: 2024-01-15.

[16] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing Idempotence for Infrastructure as Code. In *Middleware 2013 - ACM/IFIP/USENIX 14th International Middleware Conference (Lecture Notes in Computer Science)*. doi:10.1007/978-3-642-45065-5_19

[17] Patrick Tser Jern Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He, Lei Lin, Haoran Zhang, Owen M. Park, George S. Elengikal, Yuxin Kang, Ang Chen, Mosharaf Chowdhury, Myungjin Lee, and Xinyu Wang. 2024. IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs. In *Advances in Neural Inf. Processing Systems.*

[18] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. 2021. Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021 (Lecture Notes in Computer Science)*. doi:10.1007/978-3-030-72013-1_6

[19] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN 2017)*. 196–199. doi:10.1145/3092282.3092295

[20] Kief Morris. 2021. *Infrastructure as Code: Dynamic Systems for the Cloud Age* (second ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.

[21] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2023. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 534–545. doi:10.1109/MSR59073.2023.00079

[22] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. 2020. Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 238–248. doi:10.1109/SCAM51674.2020.00032

[23] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 Int. Symposium on Software Testing and Analysis*. doi:10.1145/1390630.1390635

[24] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). doi:10.1109/TSE.2019.2941681

[25] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[26] Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matthew Jones, Alessandro Morari, and Ruchir Puri. 2023. Automated Code generation for Information Technology Tasks in YAML through Large Language Models. arXiv:2305.02783 [cs.SE] https://arxiv.org/abs/2305.02783

[27] Pulumi. [n. d.]. Pulumi AI. https://www.pulumi.com/ai.

[28] Pulumi. [n. d.]. Pulumi: Infrastructure as Code in Any Programming Language. https://github.com/pulumi/pulumi. Accessed: 2024-01-15.

[29] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie A. Williams. 2019. A Systematic Mapping Study of Infrastructure as Code Research. *Inf. Softw. Technol.* 108 (2019), 65–77. doi:10.1016/j.infsof.2018.12.004

[30] Sofia Reis, Rui Abreu, Marcelo d'Amorim, and Daniel Fortunato. 2022. Leveraging Practitioners' Feedback to Improve a Security Linter. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 66:1–66:12. doi:10.1145/3551349.3560419

[31] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A Configuration Verification Tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016,*. doi:10.1145/2908080.2908083

[32] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, 189–200. doi:10.1145/2901739.2901761

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).

[34] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. 2024. Automated Infrastructure as Code Program Testing. *IEEE Transactions on Software Engineering* (2024). doi:10.1109/TSE.2024.3393070

[35] Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. 2024. A Survey of using Large Language Models for Generating Infrastructure as Code. arXiv:2404.00227 [cs.SE] https://arxiv.org/abs/2404.00227

[36] Ari Takanen, Jared D. Demott, and Charles Miller. 2018. *Fuzzing for Software Security Testing and Quality Assurance* (2nd ed.). Artech House, Inc.

[37] Michał Zalewski. 2019. American Fuzzy Lop. https://github.com/Google/AFL.

[38] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. 2019. The Fuzzing Book. https://www.fuzzingbook.org/.