Automatically Detecting Risky Scripts in Infrastructure Code

Ting Dai
IBM Research
ting.dai@ibm.com

Alexei Karve
IBM Research
karve@us.ibm.com

Grzegorz Koper
IBM GTS
grzegorz.koper@pl.ibm.com

Sai Zeng
IBM Research
saizeng@us.ibm.com

ABSTRACT

Infrastructure code supports embedded scripting languages such as Shell and PowerShell to manage the infrastructure resources and conduct life-cycle operations. Risky patterns in the embedded scripts have widespread of negative impacts across the whole infrastructure, causing disastrous consequences. In this paper, we propose an analysis framework, which can automatically extract and compose the embedded scripts from infrastructure code before detecting their risky code patterns with correlated severity levels and negative impacts. We implement SecureCode based on the proposed framework to check infrastructure code supported by Ansible, i.e., Ansible playbooks. We integrate SecureCode with the DevOp pipeline deployed in IBM cloud and test SecureCode on 45 IBM Services community repositories. Our evaluation shows that SecureCode can efficiently and effectively identify 3419 true issues with 116 false positives in minutes. Among the 3419 true issues, 1691 have high severity levels.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Availability**; • **Software and its engineering** → **Software testing and debugging**.

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421303

KEYWORDS

Infrastructure-as-Code, Ansible, Shell, PowerShell, static analysis, availability, reliability, performance, security

ACM Reference Format:
Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. 2020. Automatically Detecting Risky Scripts in Infrastructure Code. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419111.3421303

1 INTRODUCTION

Cloud computing revolutionizes the way enterprises manage their infrastructure. Cloud providers today host mission critical workloads for thousands of clients in large-scale and distributed data centers. To manage infrastructure spanning across hundreds of thousands of servers, automation is extensively used with improved productivity for infrastructure life-cycle operations such as provisioning and deployment, change management, security compliance management, and event and incident management. Such automation practices are often referred to as Infrastructure-as-Code (IaC) while the codified automation procedures are referred to as infrastructure code. In recent years, many IaC tools and techniques have been developed and adopted to facilitate the practice, such as Ansible [12], Chef [14], Puppet [16], and Terraform [19]. Ansible, a Redhat branded tool, is one of the mainstream automation tools for infrastructure provisioning and configuration management. It is widely used by enterprise clients.

Infrastructure code, such as Ansible playbook and Chef cookbook, codifies mission critical operations. Not only does it make sure infrastructure is managed for availability, reliability, and security compliance, but also it guarantees that the running applications on the infrastructure are healthy with desirable performance. Modern IaC tools have their own programming paradigm to write infrastructure code, e.g., YAML for Ansible playbooks and Ruby for Chef cookbooks. They also support embedded scripting languages such as Shell and PowerShell to either manage the infrastructure

resources like operating systems, storage and networks or interact with applications to execute the automation procedures. Risky patterns in infrastructure scripts introduce bugs and expose vulnerabilities which lead to widespread of negative impacts on availability, reliability, performance, and security across the entire infrastructure composed of hundreds of thousands of servers, causing disastrous consequences. Example risky scripts such as `Remove-Partition -DriveLetter 'C'` which removes the disk could result in business disruption when they are invoked in the production environment. Coding patterns with infinite loops can drain the computation resource, severely degrading the performance of production workloads. An unauthorized user privilege escalation or unprotected storage of credentials make the infrastructure vulnerable to cyber-attacks.

Unfortunately, different from traditional software programs like Java and C/C++ which are equipped with mature bug/vulnerability detection, testing, and verification tools [23, 24, 27, 31–33, 43], existing techniques and practices for infrastructure code and embedded scripts are rudimentary [25]. It gets more challenging when the modern code development shifts to community-based approach, where a team of contributors have mixed skills, experiences, and responsibilities. Particularly, most contributors are system administrators who lack the same level of understanding and debugging support compared with software developers [44]. Studies [2] have shown that nearly 75% of system downtime is caused by human errors. Many service outage incidents are caused by mistakes made by system administrators [1]. For example, in 2017, the Amazon S3 service became unavailable due to a removal command invoked by a system administrator who inadvertently removed a large set of servers, affecting other AWS services in the US-EAST-1 region that rely on S3 for storage, such as S3 console, EC2 new instance launches, and AWS Lambda [8]. This service disruption lasted for five hours [4] with financial loss of $150 million [5].

State-of-the-practice script checking tools, such as ShellCheck [29] and PSScriptAnalyzer [35] have brought to light the opportunity of conducting static checking over the infrastructure scripts. Those generic script-analyzers report issues in the scripts by checking their formats and syntaxes. However, without correlating the identified issues with their risky behaviors, users (e.g., system administrators) have no understanding about how the risks manifest in the production environment, what the potential business consequences of those risks have, and how severe those negative consequences are. Bridging the gap between generic script-analyzers and business consequence is one of the motivations of this work.

In this paper, we propose an automated analysis tool to identify the risky scripts in the infrastructure code, which cause availability, reliability, and performance issues or impose security vulnerabilities. Our analysis tool is static without running the target infrastructure code; thus it achieves high code coverage than the dynamic detection approach which can be interrupted by unexpected runtime errors. Moreover, our tool is platform agnostic which requires no system-specific or platform-specific knowledge, such as OS version and kernel version. To realize efficient analysis, we design a scalable analysis tool, including structured representation tree generation, script detection and composition to identify, transform and compose scripts from the original infrastructure code. The composed scripts are then checked by the open source script analyzers. The analysis results are assigned with severity levels considering business configurations and consequences. The locations of the identified issues in the composed scripts are traced back to the locations of the original source code along with the reported issues.

Our work makes the following contributions:

- We design a generic analysis framework to identify risky scripts in infrastructure code. Such product is what the current business market lacks, which has a high internal demand in IBM.
- We generate our risky code knowledge base with severity levels and business impact categories after empirically studying all 409 rules from ShellCheck and PSScriptAnalyzer.
- We implement a real-world solution based on the proposed framework, i.e., SecureCode, to check infrastructure code supported by Ansible, one of the industry mainstream automation tools.
- We integrate SecureCode with the DevOp pipeline deployed in IBM cloud and test it on 45 IBM Services community repositories. Our evaluation shows that SecureCode can identify 3419 availability, reliability, performance and security issues residing in the infrastructure code, within which 1691 have high severity levels.

The rest of the paper is organized as follows. Section 2 describes the design of our analysis framework. Section 3 presents SecureCode's implementation using our analysis framework to detect risky code in Ansible playbooks. Section 4 shows the experimental evaluation. Section 5 discusses the future work. Section 6 compares our work with related work. Finally, the paper concludes in Section 7.

2 SYSTEM DESIGN

This section discusses our design goal, followed by the overview of our analysis platform. We then describe the individual analysis modules.

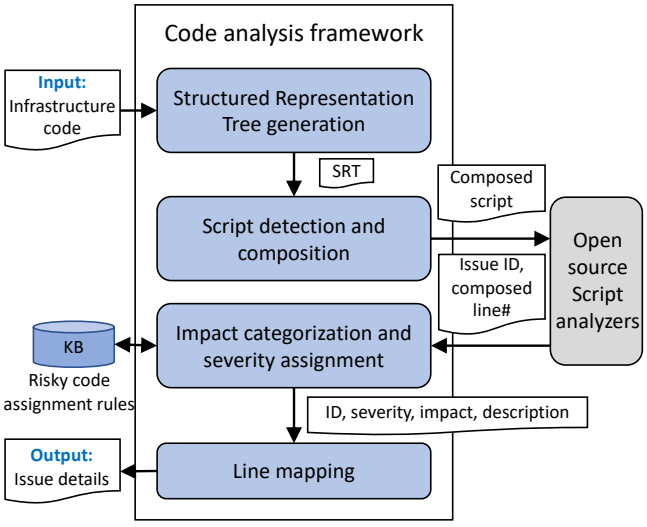

Figure 1: System architecture and program execution flow.

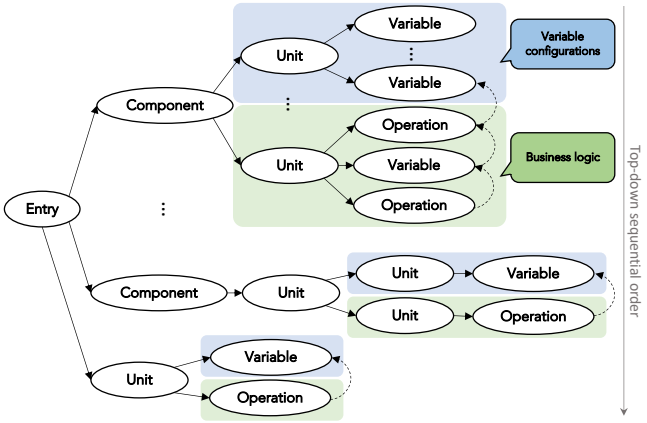

Figure 2: The structured representation tree. The "⟶" represents the containment relationship while the "- ->" represents the dependency flow.

2.1 Design Goal

Our goal is to build a generic, efficient and configurable analysis system.

Generic. Our analysis system is proposed to be generic, which is applicable to most infrastructure code. We provide programming interfaces for users to detect and compose risky scripts in different infrastructure code.

Efficient. Because of the static property, our system does not have runtime delays, e.g., task `A` can not start until task `B` finishes. Moreover, without being interrupted by unexpected runtime errors, we can achieve high code coverage.

Configurable. We support user-defined configurations to better output identified issues with respect to business clients' compliance requirements.

2.2 System Overview

To reach the aforementioned design goal, our analysis platform comes with the architecture and execution flows shown in Figure 1. Our system takes an infrastructure code repository as input. It first generates structured representation trees (SRTs) for the repository. By traversing each SRT, it composes script contents by removing template renderings after identifying that the infrastructure code invokes script executions. After passing the composed scripts to script-analyzers, it receives the checking results with issue IDs and the line numbers of the composed scripts where the issues reside. To satisfy configurable business compliance requirements, it assigns the severity level and potential impact for each identified issue based on pre-defined assignment rules retrieved from the knowledge base, and maps those issues to the corresponding infrastructure source-code lines before returning them back to the user.

2.3 Structured Representation Tree Generation

What is an SRT. An SRT is a tree data-structure to represent the syntax abstraction of the infrastructure code. It helps to extract and organize the key information in the infrastructure code and exclude details in order to ease the process of accurate script extraction. As shown by Figure 2, starting from the entry node, each SRT contains multiple components. A component consists of multiple units. A unit indicates variable configurations or represents business logic operations. A unit can also contain other units. Operations and variables are dependent on each other—an operation uses the statically configured variables, while a variable can be assigned with the value of an operation's runtime output.

Figure 2 shows that units, variables and operations are necessary in an SRT. But components are not. It is especially common when infrastructure code only contains a few automation steps and the system administrator puts all steps in a single automation file.

Top-down sequential order. All the operations in an SRT are top-down sequentially ordered. So are the units and components. For example, in Figure 2, the variable configuration unit is executed before business logic unit. When constructing an SRT, the ordering dependency between different execution steps in the same single automation file can be easily extracted—the topmost executes earliest. However, for the implicit ordering dependency, their execution sequence is defined inside of the IaC platforms rather than in the infrastructure code. The implicit ordering usually happens when referencing a statically configured variable `var` in

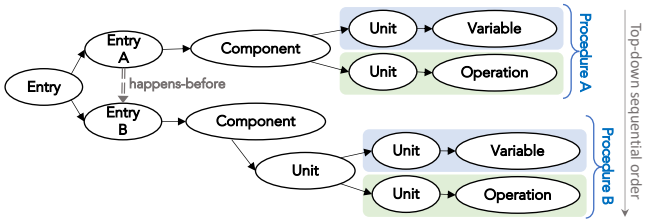

Figure 3: The happens-before relationship in infrastructure code. The "⟶" represents the containment relationship while the "= =>" represents the happens-before relationship.

an operation op. We sculpture the ordering of their (i.e., var's and op's) corresponding units by following the "reference after define" policy.

SRT forests. An infrastructure code repository contains multiple entry nodes, representing multiple automation procedures, denoted by a forest of SRTs. The ordering dependency of those SRTs is not always available or required. For example, an administrator can trigger an `install` procedure before a `delete` procedure or vice versa. However, the automation procedures' ordering does not affect the generation of their SRTs.

Merged SRTs. In other cases where two automation procedures have a happens-before relationship explicitly shown in the infrastructure code, we merge the two SRTs into one by introducing a new entry node and creating two containment flows from the new entry node to the two procedures. The first-executed procedure is on the top part of the merged SRT while the latter-executed one is on the bottom part. For example, as shown in Figure 3, $Procedure_A$ happens before $Procedure_B$, we merge them into one SRT, where the top-down sequential ordering guarantees the happens-before relationship.

When the happens-before relationships involve more than two procedures and all procedures are partially ordered, we first categorize them in different sets. Each set contains part of the procedures which can be ordered in a determined way. We then merge the procedures from the same set into an SRT. For example, we have three procedures, P_1, P_2, and P_3, which follow the happens before relationships as $P_1 \xrightarrow{\text{HB}} P_2$ and $P_1 \xrightarrow{\text{HB}} P_3$. We merge P_1 and P_2 into one SRT, and merge a duplicated P_1 and P_3 into another SRT. This also applies to the case when all procedures are totally ordered, e.g., $P_1 \xrightarrow{\text{HB}} P_2 \xrightarrow{\text{HB}} P_3$. We just need to merge all of them into the same SRT.

SRT construction. As shown in Algorithm 1, our SRT generation follows the Breadth-First Search (BFS) procedure. We first use the `getContainedObj()` function to retrieve all

Algorithm 1: SRT construction

```
input : infrastructure code repository r
output: a set of SRTs S
1  begin
2      Q ← QUEUE(), V ← ∅                          ▷ initialize the queue and visit set
3      V ← V ∪ {r}                                 ▷ add r in V
4      P ← GETCONTAINEDOBJ(r)                      ▷ get contained objects
5      for p ∈ P do
6          ENQUEUE(Q, p)
7      while Q ≠ ∅ do
8          c ← DEQUEUE(Q)
9          node_c ← PARSEOBJ(c)                    ▷ create SRT node
10         if parent[c] = ∅ then                   ▷ not contained
11             S ← S ∪ {node_c}                    ▷ add root node in S
12         N ← GETCONTAINEDOBJ(c)                  ▷ get contained objects
13         for n ∈ N do
14             if n ∉ V then
15                 ENQUEUE(Q, n)
16                 parent[n] = c
17                 children[c] ← children[c] ∪ {n}  ▷ create
                       containment relationship
18         V ← V ∪ {c}
19     O ← ORDER(S)                                ▷ get ordered lists for all objects
20     for R ⊂ O do
21         e, node_e ← EMPTYNODE()                 ▷ create empty object, node
22         for node_r ∈ R do
23             S ← S\{node_r}                      ▷ remove node from S
24             children[e] ← children[e] ∪ {r}     ▷ create
                   containment relationship
25         S ← S ∪ {node_e}                        ▷ add new root node in S
26     return S
```

the automation procedures in the root directory r (line #4) and add them in the queue (line #6). We then create a set of SRTs for all the procedures in a while loop (line #7-18). Specifically, in each iteration, we poll the head c from the queue (line #8) and use the `parseObj()` function to create an SRT node $node_c$ to abstract the syntax of object c (line #9). If object c is not contained by others, which means it is an independent automation procedure, component or unit, we consider $node_c$ as the SRT root node and add it in the tree set S (line #10-11). Next, we use the `getContainedObj()` function to extract all the contained objects in c (line #12). For each contained object n, if not visited, we add it in the queue (line #15) and create a containment relationship from object n to object c (line #16-17). In the end of the iteration, we add object c in the visit set (line #18). Our SRT generation algorithm scans the whole repository layer by layer until all automation objects (e.g., files, folders) are abstracted in an SRT node (i.e., entry, procedure, component, unit, variable and operation).

An SRT node contains a list of attribute fields, including `type`, `_file_` and `_line_`. If the node's type is *operation*, this node contains the content of the operation in the form of module-command pairs, e.g., `"shell: ls -al"`. If the node's type is *variable*, it contains the variable name with the configured value, e.g., `"user: root"`. The `_file_` attribute stores the source code file path while the `_line_` attribute stores the first line number of an operation or a variable. In an infrastructure code repository, an SRT node is uniquely identified by the ⟨`_file_`, `_line_`⟩ tuple.

To handle happens-before relationships among a forest of SRTs in S, we first extract the ordered lists [1] O for all of them (line #19). Each list R contains a sequential order among a set of SRTs, i.e., $R \subseteq S$, $R = \{srt_1, srt_2, ..., srt_n\}$, $srt_1 \rightarrow srt_2 \rightarrow ... \rightarrow srt_n$. For each list R, we use the `emptynode()` function to create a new empty object e with its corresponding SRT node $node_e$ (line #21) and add $node_e$ in the tree set S (line #26). For each SRT node $node_r$ in the ordered list R, we first remove it from the tree set S (line #23) and then create a containment relationship between $node_e$ and $node_r$ (line #24).

We should note that the interface functions, i.e., `parseObj()` and `getContainedObj()`, highlighted in Algorithm 1, should be implemented specifically to support the target infrastructure code's programing paradigm. Furthermore, the `getContainedObj()` function must guarantee that its returned values/objects are sequentially ordered—the leftmost executes earliest[2]. In such way, the sequential order is preserved among all the leaves in an SRT from left to right. The detailed implementation of the two functions will be discussed in Section 3. The other functions, including `queue()`, `enqueue()`, `dequeue()`, `order()`, `emptynode()` are commonly used library functions. We do not explicitly describe their implementation details due to page limit.

2.4 Script Detection and Composition

Script invocation in infrastructure code. Modern IaC platforms provide their own programming paradigm, syntax and libraries to support script invocations. For example, Ansible supports Shell/PowerShell script invocations in YAML playbooks using the following Ansible modules: `command`, `shell`, `script`, `raw`, `install`, `before_install`, `win_command`, and `win_shell`. Chef supports Shell/PowerShell script invocations in Ruby cookbooks using the following Chef resources: `execute`, `script`, `powershell_script`, `bash`, `csh`, and `ksh`. Puppet supports Shell/PowerShell script invocations in DSL files using the Puppet `exec` resource. Terraform supports Shell/PowerShell script invocations in TF files using the `local-exec` and `remote-exec` provisioners.

Script detection. To detect whether the infrastructure code invokes scripts, we traverse the generated SRTs, obtain their leaf nodes, and check whether the operation leaf nodes use the aforementioned script-related IaC libraries, e.g., Ansible modules, Chef resources, Puppet resources, and Terraform provisioners. As shown in Algorithm 2, our script detection module provides the `hasScript()` interface function (line #8) to check the existence of scripts in SRTs.

After detecting that the operation nodes involve the script invocation, simply dumping raw scripts from the operation nodes using the `extractScript()` interface function (line #9) is not enough, due to the template rendering in modern IaC platforms. Those raw scripts can be embedded with templated variables in Jinja2, Django, JSP, PHP, etc. To achieve accurate and modularized script checking, we need to have interfaces to compose the scripts in correct scripting language syntax and format, before passing them to the script-analyzers.

Variable map. In infrastructure code, variables are either configured in configuration files or defined by run-time jobs' execution results. The statically configured variables are in an SRT's variable leaf nodes associated with default values, while the dynamically assigned variables are in an SRT's operation leaf nodes associated with specific operations. Our system provides the `extractVar()` function to extract variable-value pairs from all variable leaf nodes and append them in the variable map M. The variable map is updated in-time—it is updated at the beginning of each loop iteration (line #6)—to guarantee the define-reference order in two aspects:

First, the variable map contains the configured variables' values, which can be retrieved and referenced in the later script composition. The sequential order in SRT generation makes sure that variable nodes are on the left side of operation nodes in the same component. Thus, the variable nodes have higher priority to be processed earlier.

Second, a dynamically defined variable in an operation node can only be referenced by the later operations. For example, there are three operation nodes op_1, op_2, and op_3 and leftmost executes earliest. If a variable var_2 is defined in op_2, the in-time updated variable map makes sure that var_2 can be referenced in op_3 but not in op_1, when we compose the scripts in those operations.

We should note that IaC tools handle lexically scope variables by overriding the old value with the new value globally, which is supported by our variable map.

[1] The ordered lists are provided by user-specific inputs or pre-requisites, defined in the Readme file.

[2] It is the implementation version of SRT's top-down sequential order.

Algorithm 2: script detection and composition

```
   input  : a set of SRTs S
   output : a set of composed scripts C
1  begin
2      for node_r ∈ S do                                   ▷ SRT root node
3          L ← ∅, M ← ∅
4          L ← getLeaves(node_r, L)
5          for l ∈ L do
6              M ← M ∪ extractVar(l)                       ▷ var map
7              if l.type = 'operation' then                ▷ operation leaf node
8                  if hasScript(l) then
9                      c ← extractScript(l)
10                     c ← composeScript(c, M)
11                     C ← C ∪ reformatTemplated(c)
12     return C
13 function getLeaves(node_r, L)
14     if children[r] = ∅ then                             ▷ leaf node
15         return {node_r}
16     for n ∈ children[r] do
17         L ← L ∪ getLeaves(node_n, L)                    ▷ union children's leaves
18     return L
19 function composeScript(c, M)
20     c' ← c
21     V ← extract(c)                                      ▷ extract the referenced variables
22     for var ∈ V do
23         if containsKey(M, var) then
24             val ← get(M, var)
25             c ← replace(c, var, val)
26     if c ≠ c' then
27         return composeScript(c, M)                      ▷ compose recursively
28     return c
```

Script composition. As shown in Algorithm 2, the script composition (line #19-28) is a process of replacing a templated variable *var* in raw scripts with its value *val* extracted from the variable map M. To handle cases with nested variable references, our `composeScript()` function recursively conducts the replacement (line #27) until all the referenced variables are replaced by their defined values or until the script cannot be further composed, i.e., the variable map M does not contain the value for a referenced variable.

For example, we have a raw shell script as `rm -rf {{dir}}`, where the `{{dir}}` variable is in Jinja2 format. The variable map contains the key-value pairs as `dir={{path}}`, `path='/'`. We call the `composeScript()` function to extract the `dir` variable (line #21), query the map (line #23), and retrieve its value as `path` (line #24). We update the script by replacing `{{dir}}` with `{{path}}` and get `rm -rf {{path}}` (line #25). A recursive function call is invoked to check whether the script can be further composed (line #27). After a second-round of extraction, query, and replacement, the script is updated as `rm -rf /`. This is the final composed script, since there is no referenced variable embedded in it.

Templated variable reformat. The composed scripts may still contain templated variables which are undefined in the infrastructure code, i.e., the variable map M does not contain the value for a referenced variable. Passing those syntactically problematic scripts to script-analyzers results in inaccuracy and false positives. To remove the templating language from the composed scripts while still preserving the variable reference, we use the `reformatTemplated()` interface function (line #11) to reformat those templated variables in correct script syntax.

We should note that the `reformatTemplated()` interface function along with others, i.e., `extractVar()`, `hasScript()`, `extractScript()`, and `extract()`, highlighted in Algorithm 2, should be implemented specifically for the target infrastructure code due to the corresponding programming paradigm. The implementation details of these functions will be discussed in Section 3. Other functions, such as `containsKey()`, `get()`, and `replace()` are commonly used library functions. Their implementation is omitted in the paper.

2.5 Business Impact Categorization and Severity Assignment

Our analysis system leverages the open-source state-of-the-practice script-analyzers such as ShellCheck and PSScriptAnalyzer to conduct risky code checking on the composed scripts. However, the issues reported by those script-analyzers are generic. They label each issue with the type of "style", "info", "warning", or "error" without clearly showcasing the potential business impacts of those issues and how severe those impacts are, when the reported issues' risky behaviors manifest in the production environment. To satisfy business compliance requirements, our analysis system regulates the identified issues with impact categorization and severity assignment.

We conduct an empirical study on the existing script-analyzers' rulesets, including 345 rules from ShellCheck and 64 rules from PSScriptAnalyzer. We study the code patterns checked by each rule to identify what risky behaviors and consequent negative impacts the problematic code can have. We categorize the issues based on the potential impacts as "none" issues, "security" vulnerabilities, "availability" issues, "performance" issues or "reliability" issues. For each category of risky code, we assign severity levels accordingly.

None risky behavior. For the problems labeled as "style" by the script-analyzers, they usually do not have risky behaviors. We mark their impacts as "none" and assign their severity levels as "low". For example, ShellCheck considers that the shell command `echo $(cat foo.txt)` has a style

issue, wherein `echo` is useless. This command does not have any risky behavior[3] but a simplified version is preferred, i.e., `cat foo.txt`.

Security. For the issues which contain security vulnerabilities, we assign their severity levels as "high", based on IBM business security requirements [6, 18]. For example, in the shell command `find . -name '*.txt' -exec sh -c 'echo "{}"' \;`, the filename is passed in by an injected shell string (i.e., `"{}"`). Any shell meta-characters in the filename can be interpreted as part of the script, which allows arbitrary code execution exploitation.

Availability. For the issues which can cause the system to be unavailable, leading to potential service outages [11], we assign their severity levels as "high". For example, the shell command `rm -rf /` deletes the whole system directory, which severely impacts system availability, causing catastrophic consequences.

Performance. For the issues which degrade system performance, we assign their severity levels as "medium". For example, the shell command `if ["$(find . | grep 'IMG[0-9]' )"]` iterates the entire directory and reads all matching lines into memory before making a decision rather than stopping at the first matching line. It prolongs the infrastructure code execution, which can decrease system performance. When a performance issue involves abnormally high CPU or memory usage, such as an infinite loop, it can significantly impact the whole system, making the system become partially or entirely unavailable. However, without runtime information and specific user input, we currently cannot decide whether the risky code has performance issues or performance-induced availability issues.

Reliability. For the issues which make the scripts' output become unreliable, further affecting infrastructure life-cycle operations, we measure them from a context specific perspective. Specifically, we extract the infrastructure operation criticality from operation names, automation file names, and comments. We classify the operations as "critical" if they are related to reboot, restart, update, backup, database, service, etc. We classify the operations as "moderate" if they are related to file read/write, word count, logs, etc. The other operations are considered as "trivial", such as printing messages on a terminal. For the unreliable scripts which are related to the infrastructure critical, moderate, or trivial operations, we consider their severity levels as "high", "medium", or "low", respectively. For example, the shell command `ps ax | grep python` is unreliable. It not only searches python processes but also tries to match against other fields, such as if the user's name was *pythonguru*. When the command's output is used in a subsequent reboot operation, all the processes created by the *pythonguru* user get restarted, affecting all the running workloads. Thus, this unreliable script's severity level should be assigned as high. However, when this shell command's output is not used or simplify printed on a terminal, this unreliable script's severity level should be assigned as low.

We generate a configurable knowledge-base to store the impact and severity information for each rule. Each rule has an ID in the format of analyzer-abbreviation and digit numbers. For the ShellCheck rules, their IDs are in the range of SC1000-SC9999, while for the PSScriptAnalyzer rules, their IDs are in the range of PS1000-PS9999. Our knowledge-base also contains short and detailed rule descriptions associated with each rule ID. A short description is a message containing both pattern and impact information to briefly summarize the risky code. A detailed discussion with examples and specific scenarios are explained in a description file. We should note that, the knowledge-base is configurable—users can easily turn on/off a rule or adjust a rule's severity level. In our evaluation, we include all the rules from the knowledge-base with the severity levels assigned by our emperical study results.

2.6 Output Format with Line Mapping

Our analysis system provides a user-friendly output for each detected issue, including ❶ a rule ID, ❷ an issue type inherited from the script-analyzers, ❸ a severity level, ❹ the potential business impact, ❺ a short description for the matched rule, ❻ a file link for the rule's detailed descriptions, ❼ the content of the original risky script with optional template rendering, ❽ the content of the composed risky script, and ❾ the source code file path with ❿ the line number where the risky script reside. Example output is discussed in Section 4.1.

The above first six fields can be retrieved from the knowledge-base. The original and composed risky code contents can be derived from our script composition module. The source code file path can be obtained from the `_file_` attribute in the operation SRT node where the script is invoked. The risky script line number can be obtained from the following formula "$L = L_0 + i - 1$ ", where L_0 is the line number of the embedded scripts stored in the `_line_` attribute of the SRT node; i is the line number of the risky code reported by script-analyzers in the composed script.

3 IMPLEMENTATION

We come up with a real-world solution, i.e., SecureCode, using our analysis platform to identify the risky scripts in Ansible infrastructure code, i.e., Ansible playbooks. To support the programming paradigm in Ansible playbooks, which are

[3]It might cause a millisecond overhead with negligible performance degradation.

written in YAML with Jinja2 template rendering, we propose our own template parser as well as reusing parsing functions in the Ansible-lint [13] tool to implement SecureCode, especially for implementing the interface functions in Section 2, including the `parseObj()` and `getContainedObj()` functions in Algorithm 1, the `extractVar()`, `hasScript()`, `extractScript()`, `reformatTemplated()`, and `extra-ct()` functions in Algorithm 2.

3.1 SRT Construction Implementation

In Ansible, a *playbook* represents an automation procedure. A *playbook* contains different *roles* as the SRT entry node contains multiple components. A *role* contains *vars*, *defaults*, and *tasks* directories as a component node contains different unit nodes. Variables are configured in the *vars* and *defaults* directories while the business logic is represented by the operations defined in the *tasks* directory.

Our `parseObj()` function's implementation takes the structured directory of Ansible playbooks into consideration, and creates the SRT entry nodes, component nodes and unit nodes accordingly. To create SRT variable nodes and operation nodes, we parse YAML files via the `parse_yaml_linenumbers()` function from Ansible-lint, and extract operation invocation statements and variable configuration statements.

We implement the `getContainedObj(c)` function to extract the containment relationships among Ansible playbooks, roles, variables, and operations in two ways. If the automation object `c` is a folder (e.g., the root directory, the *roles* directory and the *tasks* directory), the contained objects are all the files and subdirectories in this folder. Particularly, when the automation object `c` is the *roles* directory, the `getContainedObj(c)` function returns *defaults*, *vars*, and *tasks* in order, to make sure that the variable nodes are on the left side of operations nodes in an SRT. If the automation object `c` is a file (e.g., a *main.yml* file in the *tasks* directory), we extract all the contained objects by this file via the `play_children()` function from Ansible-lint. For example, if the *main.yml* file includes the *before.yml* file, then we extract *before.yml* as the contained object by *main.yml*.

3.2 Script Detection and Composition Implementation

In Ansible, every operation is conducted by specific Ansible modules or user-specified plugins. An Ansible operation is represented by a *module-parameter* pair or a *plugin-parameter* pair. For example, in the operation of `"shell: cat {{fp}}.txt"`, `shell` is an Ansible module to execute shell commands while `cat {{fp}}.txt` is the command content to be executed.

We implement the `hasScript()` function to detect scripts by checking whether the module-parameter pairs in SRT operation nodes are script related. The script-related Ansible modules include the `command`, `shell`, `script`, `raw`, `install`, `before_install`, `win_command`, and `win_shell` modules. Our `extractScript()` function then retrieves the parameter content from the script-related module-parameter pairs as the raw scripts. Currently, SecureCode does not check user customized script plugins during script detection and extraction, based on two reasons. First, the script-related modules provided by Ansible are abundant for users to execute Shell or PowerShell commands in infrastructure code. Second, user-specified shell plugins are not guaranteed to be compatible with state-of-the-practice script analyzers.

In Ansible, a statically configured variable is represented by a *variable-value* pair while the dynamically defined variable is represented by a *variable-operation* pair. We implement the `extractVar()` function to extract both static and dynamic variables. Specifically, the static variable-value pairs extracted from the variable nodes can be directly appended into the variable map M. As for the dynamic variable-operation pairs extracted from the operation nodes, we retrieve the variable's value based on the module type in the operation. If the operation contains script-related modules, such as `shell`, `win_shell`, we retrieve the command content in this operation as the variable's value. Otherwise, we simply assign a default value to this variable. In the current implementation, SecureCode does not understand all the Ansible modules, thus it cannot transform those script-unrelated operations into scripting languages.

Ansible uses Jinja2 template rendering to enable dynamic expression and access to variables. Those templated variables can be embedded in the raw scripts. We implement the `extract()` function with a template parser to extract the Jinja2 variables from the raw scripts during script composition. Our template parser contains a set of regular expressions for different matching and extraction purposes.

We first check whether the Jinja2 variables contain complex structures, i.e., filters and concatenations. We convert them into multiple simple ones via a set of well-designed regexes. For example, we convert `{{(disk _image.path | dirname) + '/' + disk_name}}` into `{{disk_image}}/{{disk_name}}`. Those simplified Jinja2 variables only contain letters, numbers, and the underscore symbol, surrounded by double braces, i.e., `{{}}`.

Then, the `extract()` function can get a list of template-free variables by removing the surrounding double braces in the simplified Jinja2 variables, e.g., `disk_images`, `disk_name`.

We implement the `reformatTemplated()` function to check whether the composed script still contains templated variables and reformat them in scripting language. Specifically, we use the `"{{[0-9A-Za-z_.{}]*}}"` regex to search the composed script and replace the double braces with the dollar symbol. For example, if the `{{file}}` variable is not defined

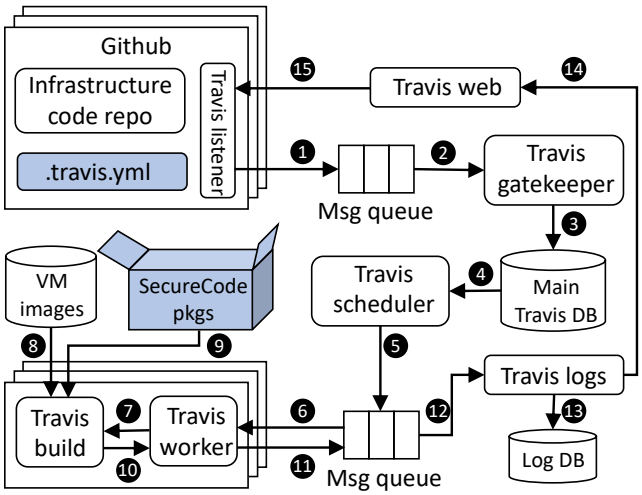

Figure 4: Travis flow. The .travis.yml file contains a Travis job configuration, including SecureCode's pre-installation, installation and execution steps.

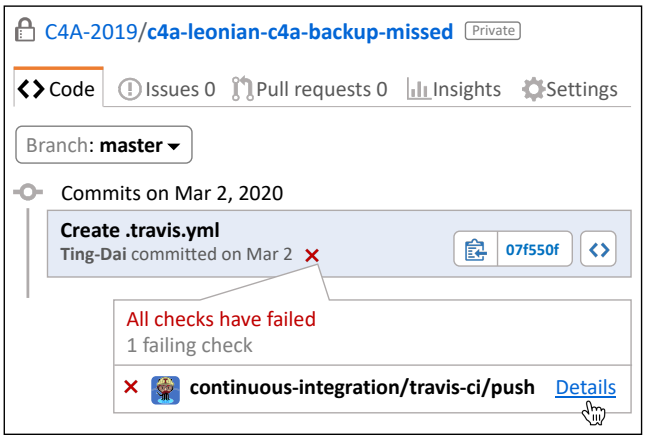

Figure 5: SecureCode's checking status for the Github commits. To view SecureCode's output, click on "Details".

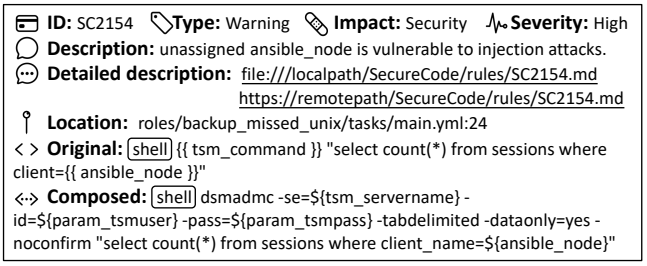

Figure 6: A snippet of SecureCode's detection report presented in Travis web GUI.

in the infrastructure code but still referenced in the shell script `cat {{file}}`, the `reformatTemplated()` function generates a syntactically correct script variable `"$file"` to replace `{{file}}` and updates the script as `cat "$file"`.

4 EVALUATION

We implement SecureCode in Python language, using parsing functions in Ansible-lint v4.2.0, Shell script analyzer ShellCheck v0.7.0, and PowerShell script analyzer PSScriptAnalyzer v1.18.3. We push SecureCode in IBM's enterprise github, integrate it with DevOp pipeline deployed in IBM Services using Travis CI [20], shown by Figure 4. We test SecureCode in 45 IBM Services community github repositories in the Call-for-Automation (C4A) contest hosted by the IBM Continuous Engineering team.

For each github repository, we create a Travis job to conduct SecureCode's checking on the automation files with the extension of `.yml`, `.yaml`, `.sh`, and `.ps1`, in the master branch. A Travis job is defined by the `.travis.yml` configuration file, which contains three steps: pre-installation, installation, and a running command to invoke SecureCode. In our experiments, all Travis jobs are running on a Ubuntu v16.04.5 VM with kernel v4.19.52, provisioned on IBM Cloud.

4.1 Output and Case Study

As shown by Figure 4, every target Github repository is associated with a webhook, i.e., Travis listener. With each new Github commit, SecureCode is invoked to check the repository code in a VM (Step ❶ to ❾). After SecureCode finishes checking, the checking status returns to the corresponding Github commit (Step ❿ to ⓯). If SecureCode identifies an issue, the commit is associated with a red cross mark (otherwise a green check mark), shown by Figure 5. The developer can click on "details" to see SecureCode's detection reports.

Figure 6 shows the output of one of the detected issues by SecureCode. The risky code happens at line #24 in the *main.yml* file, where a shell script is invoked. The original shell script contains Jinja2 template rendering variables, i.e., `{{tsm_command}}` and `{{ansible_node}}`. SecureCode conducts script composition by replacing `{{tsm_command}}` with its configured value extracted from the variable map, and then reformats the composed script by replacing double braces with the dollar symbol for those undefined Jinja2 variables, e.g., `{{ansible_node}}`. The risky pattern of undefined variables matches ShellCheck's Rule #2154 with the type of warning. SecureCode reports this issue containing security vulnerabilities with severity level as high based on our business compliances in the risky code knowledge-base. The statically unconfigured `ansible_node` variable allows a user to pass any value from a command line when he/she executes the corresponding Ansible playbooks. Without runtime sanity checking, an unexperienced user may falsely

pass wrong values to the infrastructure code. Even worse, a malicious user can inject untrusted inputs to the infrastructure, compromising the system and stealing confidential informations. In this example, `ansible_node` is used in a SQL command, which is vulnerable to SQL injection attacks. SecureCode also provides detailed descriptions about this issue with local file address and remote wiki page.

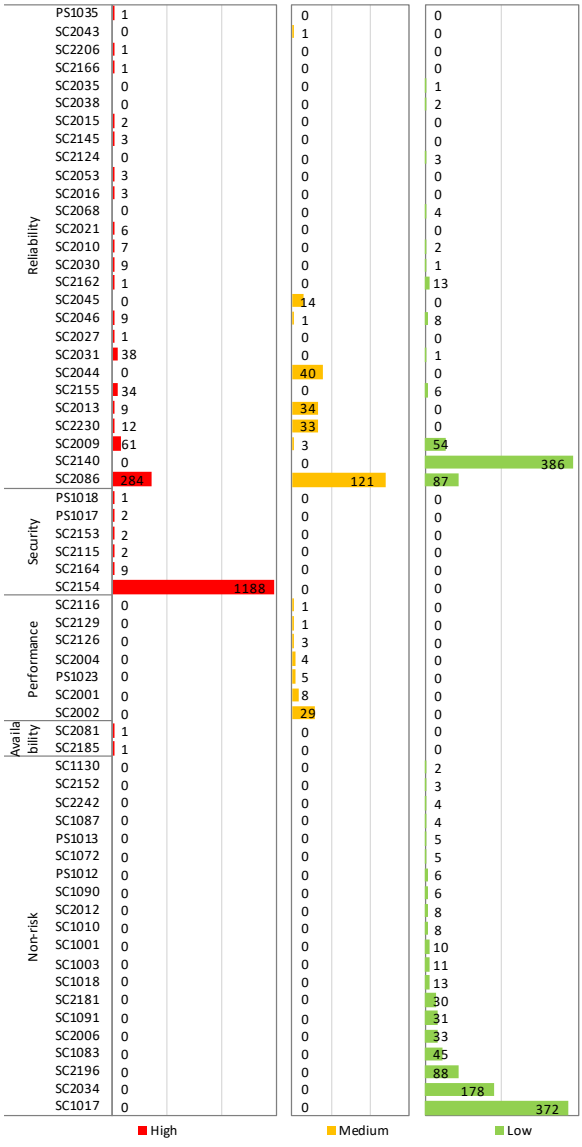

Figure 7: The statistics of identified issues with corresponding rules in each impact category with different severity levels. The y-axis represents a rule ID while the x-axis is the number of matched issues for a rule.

4.2 Detection Accuracy and Statistics

SecureCode identifies 3535 issues in total from the 45 repositories which contain 1492 automation files. We manually investigate all issues and filter out the false positives, which are 1) either falsely matched the code patterns by the corresponding rules in ShellCheck or PSScriptAnalyzer, 2) or misinterpreted by SecureCode's script composition module. SecureCode's detection is accurate with 116 out of 3535 issues as false positives.

The statistics of the 3419 true issues are shown in Figure 7. The number of matched issues for each rule varies drastically. Some rules such as SC2140, SC2086, SC2154, and SC1017 have the highest matched numbers, because their risky code patterns are related to common mistakes users make. We also have observed that one repository can have hundreds lines of code with the same risky pattern. It implies that users especially system administrators have strong coding personalities leading to the same mistakes [7]. We should notice that, there are fewer PowerShell scripts than Shell scripts in our benchmarks; thus the PSScriptAnalyzer rules match fewer issues than the ShellCheck rules, in general.

There are 1204 security issues, 485 reliability issues, and 2 availability issues having the high severity level. Particularity, the Rule SC2154 and SC2086 match most of them. The Rule SC2154 is that unassigned variable is vulnerable to injection attacks, with example discussed in Section 4.1. The Rule SC2086 is that unquoted variable/command causes globbing and word splitting, resulting in unreliable outputs. When those unreliable outputs are used in system critical operations, such as reboot, stop and install, they put the whole system in an unstable state with the potential service downtime or even outage.

SecureCode identifies 247 and 51 medium severity level issues which are related system reliability and performance, respectively. Among all the 298 medium severe issues, the Rule SC2086 match most of them when the corresponding scripts are conducting non-critical operations consuming moderate computation resources, e.g., logging in a loop. It is challenging for SecureCode to differentiate whether a loop can become infinite without specific inputs in a runtime environment. Thus, for all the loop-related issues, we currently mark their severity levels as medium.

SecureCode identifies 1430 issues with low severity level. They 1) either have none risky behaviors, such as Rule SC2034 (i.e., unused variable) and Rule PS1012 (i.e., lines end with whitespace), 2) or are related to trivial operations such as simply printing messages on a terminal.

Baseline comparison. We conduct a comparison experiment by running vanilla Ansible-lint, ShellCheck and PSScriptAnalyzer on the same 45 github repositories as SecureCode. To satisfy language capability in order to get valid

checking results, we run Ansible-lint on all the automation files with the `.yml` and `.yaml` extensions, ShellCheck on all the `.sh` automation files, and PSScriptAnalyzer on all the `.ps1` automation files. Among all the 3535 issues in the 45 repositories detected by SecureCode, 1718 can be detected by ShellCheck and 20 can be detected by PSScriptAnalyzer. The remaining 1797 issues are caused by scripts invoked inside `.yml` and `.yaml` files, which can only be detected by SecureCode. We observe that Ansible-lint can barely detect any script-related issues when they are invoked in different Ansible modules. This is because Ansible-lint is specifically designed to check the formats and best practices of YAML language. As for ShellCheck and PSScriptAnalyzer, they can detect all the issues in `.sh` and `.ps1` files as SecureCode, which is expected. This is because, unlike Shell and Powershell scripts invoked in the `.yml` and `.yaml` files, the automation procedures listed in the `.sh` and `.ps1` files do not require SecureCode's script detection and composition. SecureCode leverages ShellCheck and PSScriptAnalyzer to conduct script checking, thus the 1718 issues in the `.sh` files can be detected by both ShellCheck and SecureCode while the 20 issues in the `.ps1` files can be detected by both PSScriptAnalyzer and SecureCode.

4.3 Preparation and Detection Time

Preparation time. Before conducting detection on the target github repositories, SecureCode has a preparation stage, including pre-installation and installation. Figure 8 shows SecureCode's preparation time in each Travis job. SecureCode's average preparation time is 136 seconds, including 85 seconds of pre-installation and 51 seconds of installation.

SecureCode's preparation time does not include all the latencies from when a new Travis job is created to when the job starts to execute (Step ❶ to ❼ in Figure 4), including queuing delays, scheduling delays, database read/write delays, etc. It also does not contain the environment building delay, which refers to the time of launching a VM with default environment setups [21], such as installing python and git (Step ❽ in Figure 4). Travis is running in parallel, the aforementioned latencies (or delays) heavily depend on the workloads and are easily affected by network connections. We focus on evaluate the execution time of SecureCode not the latency of the Travis system. Thus, we do not include those delays in our evaluation.

Pre-installation time. SecureCode's average pre-installation time is 85 seconds, with 27 out of 45 Travis jobs pre-installed in 106.50 ± 5.06 seconds and 18 out of 45 jobs pre-installed in 51.54 ± 2.35 seconds. Among all the pre-installation steps, installing PSScripAnalyzer on PowerShell Core consumes the majority (i.e. 54%) of the time on average. The consumption time of this step is also highly fluctuant with the standard deviation of 27.01, which may caused by the compatibility of using Microsoft products on the Linux system.

Installation time. Triggering SecureCode's installation is easy by operating the `pip install git+https` command. The pip package manager then automatically generates the setup scripts, i.e., `setup.py` from SecureCode's `setup.cfg` file which contains the default setup configurations and commands to build a distribution. It takes the Travis VM 50.63 ± 5.37 seconds on average to install SecureCode, shown in Figure 8.

Detection time. Figure 9 shows SecureCode's detection time and lines of code (LOC) in each repository. In most cases, the detection time is proportional to the LOC with about 1 second spent on analyzing 80 LOC. However, in some repositories, such as $Repo_3$, $Repo_9$, $Repo_{15}$, $Repo_{18}$, $Repo_{25}$, and $Repo_{37}$, the ratio of detection time to LOC is much lower than others. This is because SecureCode's detection time is not only related to LOC, but also related to the amount of script invocations and the complexity of script composition in each repository. We observe that in $Repo_3$, $Repo_{15}$, and $Repo_{18}$, there are fewer script invocations even they have a large number of LOC. Meanwhile, in $Repo_9$, $Repo_{25}$, and $Repo_{37}$, they invoke simple scripts which do not require SecureCode's comprehensive composition step. On the contrary, for repositories containing complicated script invocations, such as $Repo_{12}$ and $Repo_{44}$, SecureCode spends more time on script composition; thus the ratio of detection time to LOC in these repositories is higher than others.

4.4 Coding Skill Variation

We use both 1) the number of detected issues and 2) the number of detected issues per LOC (i.e., *ipl* ratio) to evaluate the code quality of each tested repository. As shown in Figure 10, the issue numbers and *ipl* ratios tend to fluctuate utterly, ranging from 0 to 1414, and 0 to 0.45, respectively. The average and median issue numbers are 79 and 6; while the average and median *ipl* ratios are 0.04 and 0.01. Repositories have more (severe) issues or higher *ipl* ratios tend to have lower quality of code. For example, $Repo_1$ and $Repo_{34}$ are considered to have lower code quality with a large number of issues (545 and 1414, respectively, $\gg 79 > 6$) and higher *ipl* ratios (0.36 and 0.29, respectively, $\gg 0.04 > 0.01$). We also observe that most of issues in these repositories have high or medium severity levels. Another examples are $Repo_2$ and $Repo_9$. They have lower code quality with large numbers of issues (691 and 285, respectively, $\gg 79 > 6$), among which most have high severity levels, even with average *ipl* ratios, i.e., 0.04. On the contrary, repositories have less (severe) issues and lower *ipl* ratios tend to have higher quality of code, e.g., $Repo_{11}$, $Repo_{14}$, $Repo_{16}$.

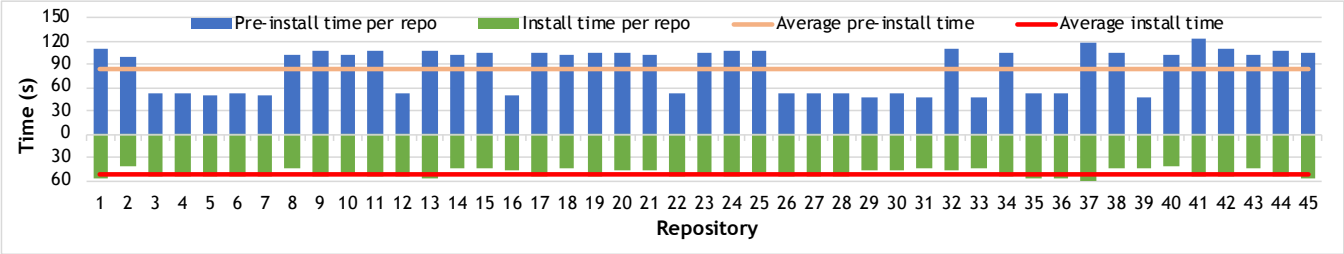

Figure 8: SecureCode's pre-installation and installation time.

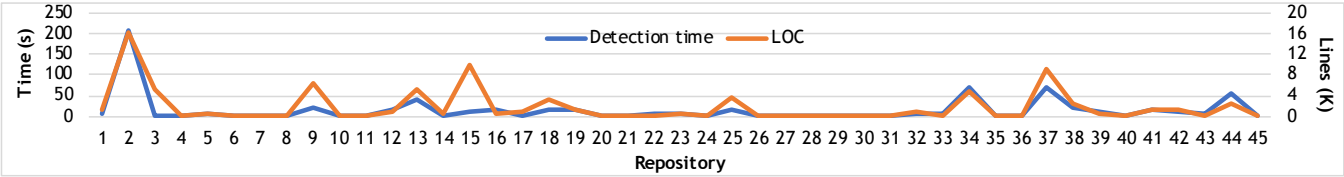

Figure 9: SecureCode's detection time and LOC in each repository.

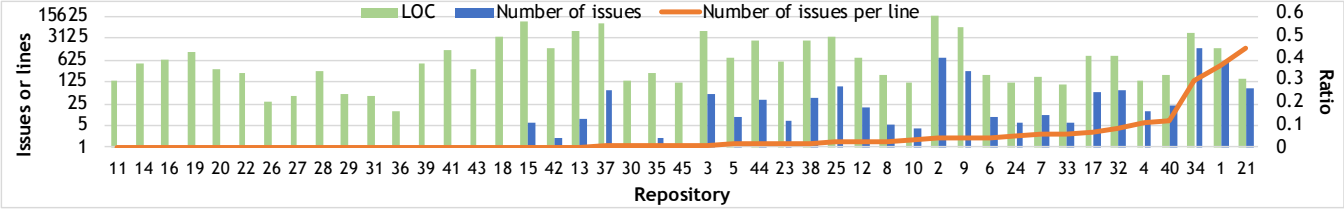

Figure 10: LOC, the number of detected issues, and their ratios in each repository. The ratios are in an ascending order.

The variation of code qualities are caused by different development experience, coding practice, and scripting language expertise. Especially for the C4A contest, not all participants are fully trained, nor are they full-time dedicated on the project for the contest. SecureCode analyzes the submitted infrastructure code and checks whether they contain risky patterns on script invocations. However, judging the submitted code based on SecureCode's detection results is far from enough. Moreover, we never depreciate those repositories with large number of detected issues. They may propose new features and solutions to the Ansible infrastructure, which can be widely used and adopted in future after some refinement and refactoring.

4.5 User Experience

We have designed quantified metrics to access the user experience from IBM Continuous Engineering team by comparing SecureCode with manual checking [4], ShellCheck [5] and PSScriptAnalayzer [6] in terms of throughput improvement (i.e., LOCs reviewed per person per day) and efficiency improvement (i.e., number of issues to be identified). The pre-production pilot of SecureCode comes back with the feedback that SecureCode achieves an estimated 5x throughput improvement compared with manual check, 2x to 5x throughput improvement compared with ShellCheck, and 2x to 3x throughput improvement compared with PSScriptAnalzyer. SecureCode can identify issues which were overlooked by developers. Specifically, SecureCode can identify 5x more issues than manual check, 2x to 3x more issues than ShellCheck, and 2x to 3x more issues than PSScriptAnalyzer. This user experience matches our baseline comparison results, which showing that SecureCode removes the bottleneck of code governance/review and shortens the time to market.

[4]Manual checking means checking the Ansible-embedded Shell/Powershell scripts manually without any tool.

[5]Users can compose their own Shell scripts from Ansible playbooks and then run those composed scripts using ShellCheck.

[6]Users can compose their own Powershell scripts from Ansible playbooks and then run those composed scripts using PSScriptAnalyzer.

5 FUTURE WORK

In this work, we take a first step towards the direction by identifying risky scripts in infrastructure code. The fundamental idea is to bridge the gap between generic state-of-the-practice script-analyzers and business compliances when detecting risky patterns with correlated potential negative consequences in the business infrastructure environment.

SecureCode is a roadmap in this journey. Additional features to be implemented are on our schedule, including supporting more scripting languages, e.g., perl, python, and ruby, and supporting checking infrastructure code in other IaC tools, e.g., Chef cookbooks and Puppet manifests. To support other scripting languages, we need to integrate corresponding script analyzers into our system and one-time empirically study on the out-of-the-box rulesets, e.g., Bandit [37] for Python. To support other IaC tools, specific implementations on the interface functions in Algorithm 1 and Algorithm 2 are required. With the help of open-source libraries, e.g., puppet-strings [17], the implementation work is not a heavy load.

Our knowledge-base has a good coverage on common syntax and intermediate-level semantic bugs. In our pre-production testing, we do not encounter any false negatives. However, SecureCode cannot detect certain types of bugs, e.g., dirty copy-on-write, and fork bomb. Thus, extending the rulesets of SecureCode is also our future work.

6 RELATED WORK

Infrastructure code analysis. Previous work has been extensively done to identify quality concerns in infrastructure code [13, 15, 22, 40, 42]. Those quality analysis tools mainly check style issues, e.g., complex expressions, long statements, and unnamed tasks. However, these quality concerns mostly do not cause risky behaviors manifested in the production environment.

Many infrastructure code analysis approaches have been proposed to identify security, availability and reliability issues. SLIC [38] detects seven security smells in Puppet manifests, including admin by default, hard-coded secret, suspicious comments, etc. FSMoVe [41] identifies ordering violations and missing notifiers in Puppet programs, which can cause the infrastructure become unavailable [3] or unreliable [39]. Model-based testing [26, 30] is proposed to detect non-idempotent and non-convergent automations in Chef recipes and Puppet manifests. Infrastructure code which cannot idempotently converges to a desired state causes reliability issues.

In comparison, our work checks the risky patterns (including availability, reliability, security and performance issues) in Shell and PowerShell scripts embedded in infrastructure code. We believe our work is complementary to the existing infrastructure code analysis approaches.

Shell and PowerShell script analysis. Recent work has been done to identify bugs and security vulnerabilities in Shell and PowerShell scripts. ABash [34] statically checks common expansion bugs in bash scripts using taint tracking and abstract interpretation. Shill [36] enforces the scripts' access control to system resources via declarative security policies and capability-based sandboxes. Machine Learning based detectors [9, 10, 28] are proposed to detect malicious PowerShell commands, using NLP, CNNs, and LSTM.

Different from existing Shell and PowerShell script checking approaches, which either are specific to certain types of bugs and vulnerabilities [34, 36] or require a large number of training data [9, 10, 28], our work leverages static pattern-based script-checking tools, i.e., ShellCheck [29] and PSScriptAnalyzer [35], to analyze infrastructure embedded Shell and PowerShell scripts and correlates them with business configurations and consequences.

7 CONCLUSION

Risky scripts in infrastructure code have widespread of negative business impacts, leading to disastrous consequences. To identify the risky scripts which cause availability, reliability, security and performance issues, we design an analysis framework to abstract the structured representations from infrastructure code before extracting and composing its embedded Shell and PowerShell scripts. Our analysis framework then leverages generic state-of-the-practice script analyzers, ShellCheck and PSScriptAnalyzer, to conduct script checking, and a configurable knowledge base to categorize business impacts and assign severity levels for the identified issues. We implement SecureCode based on our analysis framework to check Ansible playbooks. We integrate SecureCode with the DevOp pipeline deployed in IBM cloud and test SecureCode on 45 IBM Services community repositories. Our evaluation shows that SecureCode can identify 3419 true issues with 116 false positives, with the average pre-installation and installation time of 136 seconds, and average detection time of 1 second per 80 LOC. Among the identified 3419 true positives, 1691 have high severity levels which make the system become unavailable, unreliable, or vulnerable to security attacks. We also observe that the ratio of issue numbers to LOC varies drastically in each repository, indicating the community's variegated development experience, coding practice, and scripting language expertise.

REFERENCES

[1] 2012. *Critical Cloud Computing: A CIIP perspective on cloud computing services.* https://resilience.enisa.europa.eu/cloud-security-and-resilience/publications/critical-cloud-computing

[2] 2013. *Human error most likely cause of datacentre downtime, finds study.* https://www.computerweekly.com/news/2240179651/Human-error-most-likely-cause-of-datacentre-downtime-finds-study

[3] 2014. *DNS Outage Post Mortem.* https://github.blog/2014-01-18-dns-outage-post-mortem/

[4] 2017. *Amazon And The $150 Million Typo.* https://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo

[5] 2017. *AWS's S3 outage was so bad Amazon couldn't get into its own dashboard to warn the world.* https://www.theregister.co.uk/2017/03/01/aws_s3_outage/

[6] 2017. *Cyber Security Is A Business Risk, Not Just An IT Problem.* https://www.forbes.com/sites/edelmantechnology/2017/10/11/cyber-security-is-a-business-risk-not-just-an-it-problem

[7] 2017. *The Main Difference Between Junior Programmers And Senior Programmers.* https://www.forbes.com/sites/quora/2017/12/05/the-main-difference-between-junior-programmers-and-senior-programmers/#724b335e67f3

[8] 2017. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region.* https://aws.amazon.com/message/41926/

[9] 2018. *Malicious PowerShell Detection via Machine Learning.* https://www.fireeye.com/blog/threat-research/2018/07/malicious-powershell-detection-via-machine-learning.html

[10] 2019. *Deep learning rises: New methods for detecting malicious PowerShell.* https://www.microsoft.com/security/blog/2019/09/03/deep-learning-rises-new-methods-for-detecting-malicious-powershell/

[11] 2019. *A shell script that deleted a database, and how ShellCheck could have helped.* https://www.vidarholen.net/contents/blog/?p=746

[12] 2020. *Ansible is Simple IT Automation.* https://www.ansible.com/

[13] 2020. *Ansible-lint: Best practices checker for Ansible.* https://github.com/ansible/ansible-lint/

[14] 2020. *Chef: Deploy new code faster and more frequently.* https://www.chef.io/

[15] 2020. *Puppet-lint: Check that your Puppet manifests conform to the style guide.* https://github.com/rodjek/puppet-lint

[16] 2020. *Puppet: Powerful infrastructure automation and delivery.* https://puppet.com/

[17] 2020. *puppet-strings.* https://rubygems.org/gems/puppet-strings

[18] 2020. *Secure infrastructure from IBM lets you make sure your data stays safe — wherever it goes.* https://www.ibm.com/it-infrastructure/solutions/security

[19] 2020. *Terraform: Use Infrastructure as Code to provision and manage any cloud, infrastructure, or service.* https://www.terraform.io/

[20] 2020. *Travis CI.* https://travis-ci.org/

[21] 2020. *The Xenial Build Environment.* https://docs.travis-ci.com/user/reference/xenial/

[22] 2020. *YAML Lint: A linter for YAML files.* https://github.com/adrienverge/yamllint

[23] Ting Dai, Daniel Joseph Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. 2019. Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures. *IEEE Trans. Parallel Distrib. Syst.* 30, 1 (2019), 107–118.

[24] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18).*

[25] Michael Greenberg and Austin J. Blatt. 2019. Executable Formal Semantics for the POSIX Shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (2019), 30 pages.

[26] 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'16).*

[27] Jingzhu He, Ting Dai, and Xiaohui Gu. 2018. TScope: Automatic Timeout Bug Identification for Server Systems. In *2018 IEEE International Conference on Autonomic Computing, ICAC 2018, Trento, Italy, September 3-7, 2018.* 1–10.

[28] Danny Hendler, Shay Kels, and Amir Rubin. 2018. Detecting Malicious PowerShell Commands Using Deep Neural Networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS'18).*

[29] Vidar Holen. 2020. *ShellCheck – shell script analysis tool, infrastructure, or service.* https://www.shellcheck.net/

[30] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing Idempotence for Infrastructure as Code. In *ACM/IFIP International Middleware Conference (Middleware'13).*

[31] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19).*

[32] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18).*

[33] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium.* USENIX Association, USA.

[34] Karl Mazurak. 2007. ABash: Finding bugs in bash scripts. In *In ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'07).*

[35] Microsoft. 2020. *PSScriptAnalyzer.* https://github.com/PowerShell/PSScriptAnalyzer

[36] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. SHILL: A Secure Shell Scripting Language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14).*

[37] PyCQA. 2020. *Bandit is a tool designed to find common security issues in Python code.* https://github.com/PyCQA/bandit

[38] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19).*

[39] 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'16).*

[40] 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'16).*

[41] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical Fault Detection in Puppet Programs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20).*

[42] E. van der Bent, J. Hage, J. Visser, and G. Gousios. 2018. How good is your puppet? An empirically defined and validated quality model for puppet. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18).*

[43] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16).*

[44] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4, Article 70 (July 2015), 41 pages.