RESEARCH-ARTICLE

CoZure: Context Free Grammar Co-Pilot Tool for Finding New Lateral Movements in Azure Active Directory

ABDULLAHI CHOWDHURY, The University of Adelaide, Adelaide, SA, Australia

HUNG X NGUYEN, The University of Adelaide, Adelaide, SA, Australia

Published: 16 October 2023

RAID 2023: The 26th International Symposium on Research in Attacks, Intrusions and Defenses
October 16 - 18, 2023
Hong Kong, China

RAID '23: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (October 2023)
https://doi.org/10.1145/3607199.3607228
ISBN: 9798400707650

CoZure: Context Free Grammar Co-Pilot Tool for Finding New Lateral Movements in Azure Active Directory

Abdullahi Chowdhury
The University of Adelaide
Adelaide, Australia
abdullahi.chowdhury@adelaide.edu.au

Hung Nguyen
The University of Adelaide
Adelaide, Australia
hung.nguyen@adelaide.edu.au

ABSTRACT

Securing cloud environments such as Microsoft Azure cloud is challenging and vulnerabilities due to misconfigurations, especially with user roles assignment, are common. There have been significant efforts to find vulnerabilities that enable lateral movements in Azure AD systems. All of the existing works, however, either follow a manual process to find new vulnerabilities or are only able to discover whether known vulnerabilities exist in a deployed Azure environment. We develop an Azure Active Directory (AAD) lateral movement-discovery tool, CoZure, that can help researchers find new lateral movements in an Azure AD environment. CoZure deploys algorithms from Context-Free Grammar (CFG) to first learn the ways (grammar rules) that security researchers find vulnerabilities and then extend these rules to discover new lateral movement paths. CoZure first collects a large set of existing AAD environment commands using a specialized scraping tool, it then uses CFG to build a knowledge base dataset from these commands and previous attacks. Cozure then applies the knowledge learned to find new combinations of commands that could open up new candidate lateral movements, which are then tested in a real AD environment for validation and manually checked by the user. CoZure helped discover lateral movements that current fuzzing tools (e.g., OneFuzz, RESTler) cannot identify and also shows better performance in finding existing misconfiguration issues in Azure AD. Using CoZure, we have discovered two new (not previously known) lateral movement methods that could lead to numerous new attacking paths in Azure AD.

CCS CONCEPTS

• **Security and privacy** → **Vulnerability scanners**; • **Computer systems organization** → *Cloud computing*; • **Networks** → *Network security*; • **Theory of computation** → **Grammars and context-free languages**.

KEYWORDS

Azure Active Directory, Vulnerability, Cloud Security

RAID '23, October 16–18, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0765-0/23/10…$15.00
https://doi.org/10.1145/3607199.3607228

ACM Reference Format:
Abdullahi Chowdhury and Hung Nguyen. 2023. CoZure: Context Free Grammar Co-Pilot Tool for Finding New Lateral Movements in Azure Active Directory. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3607199.3607228

1 INTRODUCTION

Microsoft Active Directory (AD) is the default security management system for Windows domain networks and is a popular target for hackers [4, 31]. Azure Active Directory (Azure AD) is a recent extension of AD as a cloud service. It was launched back in 2008 and has gone through many iterations. However, the same issues that plagued AD have been found appearing in Azure AD [1, 18]. These vulnerabilities often arise from network misconfigurations that are too common when concepts such as least privilege and tiered administration are difficult to implement practically [30].

Securing Azure AD is a major challenge, and there has been a significant effort to automate the process. One of the current methods is to download a network layout and then enumerate relations between nodes to discover methods of escalating privileges. Tools such as AzureHound [24] can find misconfigurations and *identity snowball attacks* [14] and display them in an attack graph.

In this graph, a directed edge from node A to B represents that an attacker can reach from A to B via a lateral move. Attackers use this graph to navigate their way from a low-privilege user account to high-privilege accounts and ultimately, the Domain Admin. AzureHound and other existing attack graph generators [24, 25] and penetration tools such as OneFuzz [19] and RESTler [5], however, focus primarily on finding whether a known lateral movement path exists in a given AD system and cannot detect and find new or unknown lateral movements. On the other hand, security researchers, network admins, and Azure AD penetration testers manually invest many human hours in researching new possible attack paths. Finding new lateral movements is both costly and is currently done by experts who are well-versed in the art of active directory security. The current manual process is not ideal for hardening and securing fast-moving cloud environments. Our main objective in this paper is to develop new methods for discovering lateral movements in Azure AD.

Our main contribution is the development of Cozure, a co-pilot tool, for discovering new lateral movements in Azure AD. Our key innovation is to use a human language grammar model to extract rules from existing lateral movement techniques and generate new movements by extending these rules. We first observe that in most Azure AD snowball attacks, a set of commands are combined in certain patterns to create new actions that push the AD system to new, unexpected states. The set of commands is readily available

in technical specifications and their successful combinations can be extracted from existing snowball attacks. Note, however, that a random combination of commands rarely yields a successful attack. Combining these commands into successful attack activities requires expert human knowledge and is often done by security researchers with in-depth knowledge and innate insight into what combination works. Our main insight is that the manual process of creating a new attack in Azure AD resembles how human languages are learned and that this process can be modeled by a well-known grammar called *Context Free Grammar* (CFG).

A grammar represents a (possibly infinite) set of patterns using a finite lexicon and a finite set of rules or constraints that determine permissible combinations of components in the lexicon.

In the context of Azure AD, the lexicon can be thought of as the set of objects and attributes that define the identity and access management capabilities of the service. This includes things like users, groups, applications, roles, and permissions, as well as attributes like names, email addresses, and security settings. Grammar is a set of rules that govern the way in which those words or symbols can be combined to form valid expressions or statements. Grammar can also incorporate a set of testable hypotheses that serve as the theoretical foundation for explaining the observed patterns. CoZure uses two essential features of CFG named parsing (dividing the string into its component parts and describing their syntactic roles) and derivation (forming new words from existing words) to analyze existing attacks and find new lateral movements in Azure AD. The first feature (parsing) allows CoZure to learn from existing movements. The second feature (derivation) enables CoZure to find new attack methods.

Our key contributions in applying CFG to find new lateral movements in Azure AD include:

- We apply the Context Free Grammar model from natural languages to build a lateral movement discovery tool that can learn from human experts the different ways they generate new attack paths in Azure AD. The tool can then generate new movements automatically using the learned rules. To the best of our knowledge, this is the first time CFG is used to help find new lateral movements.
- Within our tool, we develop a novel parsing tool that generates a list of Azure AD CLI and PowerShell commands as input into our model. We also develop innovative left and right derivations and parsing algorithms to combine the Azure rules to reduce the number of attempts to find possible vulnerabilities in AAD.
- Our tool found two new lateral movement methods in the Azure AD environment. The first method is based on new RDP tunneling techniques, and the second method exploits open ports (typically left open by default) in Azure AD. Furthermore, our tool is more efficient than existing solutions in finding and automating existing attacks. Compared to standard fuzzers; our tool uses 5% to 40% fewer tries before successfully reproducing an attack.

The organization of the paper is as follows. The first part (Section 2) gives a brief overview of the state-of-the-art tools and techniques for identifying lateral movements in Azure AD and discusses the use of grammar, derivation methods, and parsing algorithms. Section 3 describes the details of our proposed solution. In the last part (Sections 4.2 and 4.3), we present our approach to automate existing attacks and discover new attacks by applying CFG with different parsing algorithms. We conclude with a discussion of potential research directions for using the error logs to investigate finding possible new attack edges in Section 5.

2 BACKGROUND AND RELATED WORK

2.1 Lateral movements and attack paths in Azure AD

Azure AD environments are vulnerable to identity attack paths (also known as an identity snowball attack) [14]. Typically, a snowball identity attack starts when an attacker gains initial access by compromising a low-privilege user account. The attacker then moves from low-privilege accounts to higher-privilege accounts by moving through different hosts and services and performing a set of exploits. These attacks stop when the attacker compromises the domain admin - the highest privileged account. Figure 1 shows an example of one step in the snowball attack chain in an Azure AD environment. In this example, User 1 is the owner of the resource group R1. Linux VM, Azure storage, keyVault file, and WindowsVM are part of R1. Even though User 1 is the owner of R1 and the root user of Linux VM, it does not have direct access to WindowsVM and does not have plain text read access to the KeyVault file. Thus User 1 is not allowed to read the content of the KeyVault file. An attacker having access to User 1, which has low privilege to KeyVault file but high privilege to R1 and Linux VM, won't be able to access the KeyVault by default. In these cases, the attacker will need to perform a set of actions as shown in Section 2.2 to perform a lateral movement. Each of these movements will then open up an exponential number of new attack paths (snowball attacks). A typical Azure AD can have millions of snowball attack paths [31]. The key challenge in Azure AD security is to find and eliminate potential lateral movements in order to reduce the number of attack paths.

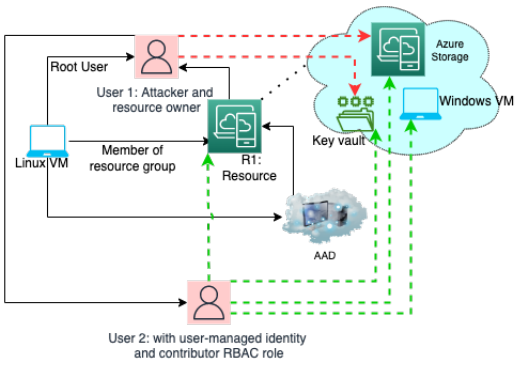

Figure 1: An example of a sophisticated lateral movement in Azure AD.

Microsoft has developed two fuzzing tools called OneFuzz [19] and RESTler [5] to find whether an existing known attack technique is applicable to an Azure AD environment. Attackers and

defenders also use other tools such as Bloodhound [26] to identify and analyse the attack paths of Azure Active Directory. PowerShell script-based tool called BloodHound Attack Research Kit[25] is used by BloodHound team to analyze the attack primitives on Azure AD environment. This tool has different predefined functions (e.g., Token Management and Manipulation, Enumeration, Meta) to assist network admins and researchers in investigating the Azure AD for any existing lateral movement path. All these fuzzers, Bloodhound, BARK work with predefined instructions and can detect if an existing system setup has any known vulnerabilities or flaws. Hackers and security researchers, however, exploit novel combinations of commands and activities to modify different Azure AD services and enable the compromised account to make new lateral movements. None of the current tools can learn these tactics to generate new attacks.

2.2 Manual Lateral Movements Discovery in Azure AD

Security researchers constantly discover new lateral movements by cleverly combining legitimate commands in unexpected ways. For example, in a blog post [16], the author shows that poor permission delegations on the Subscription or Resource Group level for System-Assigned Managed Identity can be exploited for lateral movements. The author illustrates that an attacker with owner permission of a resource group and root permission in a Linux machine can perform the following seven separate steps, combined in a sophisticated way, for lateral movements:

(1) The attacker first creates a new Windows VM with a user-assigned managed identity and adds the VM to the resource group.
(2) The attacker next executes the KeyVault policy modification command to modify the right to give the new user permission to get access plaintext version of the KeyVault (essentially exploiting the owner right of the resource group, the user-assigned identity, and the contributor RBAC role).
(3) The attacker then creates a new Windows VM using the resource owner privilege and adds the VM to the resource group. The Windows VM is assigned to the user-assigned managed identity created by the attacker.
(4) The attacker next uses Managed Identity of the Azure VM to access an Azure Key Vault by giving it the specific RBAC role (contributor).
(5) The attacker executes the KeyVault policy modification command to modify the right to give the new user permission to get the list of the KeyVault (essentially exploiting the owner right of the resource group, the user-assigned identity, and the contributor RBAC role).
(6) Having access to proper delegation within the services in the resource group, the attacker can now find details of other users details from the storage keyfile. From there, the attacker can create a new local user account with a contributor RBAC role to remain persistent.
(7) The attacker obtained Network Security Group (NSG) details to add user-assigned identities with contributor RBAC roles. The attacker can find open RDP ports for all sources from this command. Even if the RDP port is closed, the attacker can modify the NSG security policy and move laterally to another Windows machine using the RDP connection.

Similar sophisticated attacks have been demonstrated in [10, 11, 17]. Each of these requires the attacker to carry out a precise sequence of sophisticated actions to add unexpected rights or access privileges to the current user set-up. As mentioned previously, existing attack graph generation tools (BloodHound [26]), fuzzers (OneFuzz [19], Restlr [5]) work with only existing set-up and cannot be used to generate the attacks described in this section.

Our key innovation in this work is to train machine algorithms to learn and find these sophisticated attacks consisting of sequences of actions automatically. Please note that the training phase requires some manual inputs (e.g., update details of any new attacks).

2.3 Context Free Grammar for Azure AD

A good model for analyzing sequential data similar to the sequence of actions in an Azure lateral movement is a natural language model, especially grammar. The alphabet of a formal language is made up of symbols, characters, or tokens that combine to form the language's strings. Context-free grammar [13] can be used to create languages devoid of context. They do it by taking a collection of variables that are recursively specified in terms of one another by a set of production rules. Formally, a context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple $(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$, where $\mathcal{T}$ is the alphabet (the set of terminal symbols), a (finite) set of states $\mathcal{N}$ (the set of non-terminal symbols), starting symbol $\mathcal{S}$, and a (finite) set of production rules $\mathcal{P}$. $\mathcal{P} : \mathcal{N} \rightarrow (\mathcal{N} \cup \mathcal{T})^*$. In this definition of P, the $*$ operator represents the Kleene star operation [27].

Context Free Grammar has been used in different fields related to anomaly, code vulnerabilities, and pattern recognition [22]. The method of processing and analyzing strings is referred to as string manipulation. It entails several operations involving altering and parsing strings to use and change their data. String manipulation problems are one of the most common causes of security flaws or anomalies. Cross-Site Scripting, Injection Flaws (e.g., SQL injection), and Malicious File Execution are the top three vulnerabilities in the Open Web Application Security Project's top ten lists [21], all of which are caused by incorrect input string sanitization and manipulation.

Code vulnerabilities can result from string manipulation errors involving harmful elements, potentially leading to regrettable damage [2]. Recent research demonstrates progress in various estimation methods for string evaluation. Abstract interpretation is also employed to establish accuracy using stringent mathematical strategies. Abstract interpretation is used in Control-Flow Graph (CG) and Data-Flow Graph (DG) to detect anomalies in different platforms [32][9][23][8].

Grammar-based models *have never been used* for analyzing attacks on Azure AD - the focus of this paper. Implementing CFG in Azure AD requires the development of complex and dynamic grammar rules that are compatible with Azure AD services and associated third-party applications within Azure AD. Our main modifications to existing CFG models and techniques for Azure AD are as follows:

(1) To capture the nuances of various security policies, including role-based access control, attribute-based access control, and dynamic access control mechanisms, we use bespoke grammar specifically designed for the Azure AD domain. We modified the Context Free Grammar so that it can analyze the syntax of the commands used in Azure AD and parse the commands using our proposed parsing algorithm.
(2) Standard parsing algorithms in CFGs do not work directly on Azure AD data, especially in handling the ambiguity issue that is common in Azure attacks. Ambiguity occurs when a grammar permits multiple valid parse trees for a given input, which can complicate the correct interpretation selection and may necessitate additional processing or disambiguation steps for resolution. Cozure documents this supplementary process and all possible trees generated from this ambiguity to discover new lateral movement paths.
(3) Cozure's parsing algorithm is specifically designed to handle commands used in the Azure AD environment (CLI and PowerShell). We created two separate sets of command lists for both CLI and PowerShell so that our model can work well in both environments.

3 ATTACK PATH DISCOVERY MODEL AND ALGORITHMS

We present here our CFG model for Azure AD lateral movements. The model is trained and used in two separate steps. In the first step a set of known lateral movements, often demonstrated and published online by a researcher or Whitehat hacker, is used to train the model in the rules that security researchers use to create new lateral movements. The second and most important step is to explore new lateral movements in the Azure Active Directory (Azure AD) using the learned rules and syntax of a CFG grammar.

3.1 Key Entities in Our Model

There are four key entities in our model: *command*, *activity*, *security attribute*, and *target*.

In our model, *commands* are the smallest building blocks. These are Azure commands such as *az group list*. An *activity* is an ordered subset of commands that are combined together to achieve an objective. Note that not all arbitrary subsets of commands form an activity, only those when combined together in a given order results in useful output information such as the list of users. In our model, the set of activities is constructed by extracting activities listed in existing lateral movements or technical specifications. An example of Azure AD activity is "GetUserInfo". This activity will include commands for get User ID *(az ad user show - -id <user_principal_name> - -query objectId)*, Tenant ID *(az account show - -query tenantId)*, get user AAD roles *(az role assignment list - -assignee <user_principal_name>)*, and RBAC roles *(az role assignment list - -scope <resource_id> - -assignee <user_principal_name>)*.

A security attribute is a set of multiple activities. The security attribute for Keyvault file includes multiple activities such as Tenant-Details, ListedApplications, and Authentication method. A *target*, such as an Azure AD, is a set of multiple security attributes. For example, if the target is Azure AD storage, the security attributes will be KeyVault, Blob, FileStorage, SecurityPolicies, and so on.

Figure 2 depicts the key components in our model and how they relate. For example, the target Azure environment contains 4 security attributes (Azure Storage, Blob, KeyVault, File Storage, SecurityPolicy) in the picture. Notes there are other attributes not shown in the figure. One of the security attributes is KeyVault. The KeyVault security attribute shown here contains two activities: (1) getting a user account and (2) obtaining a certain credential for the user. The activity "User Details" consists of three commands:

```
az ad user show --id <user_principal_name>
--query objectId
az ad user show --id <user_principal_name>
az role assignment list --assignee
<user_principal_name>
```

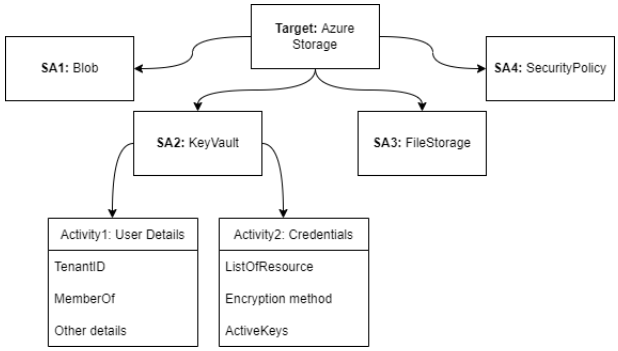

Figure 2: Activities, Security Attributes, and Target of Azure AD. SA stands for security attributes.

3.2 Building AAD Data Sets for the CFG model

3.2.1 Building the Command Set for AAD . The first step in constructing the CFG model is to build the set of commands. The command set contains commands in Azure AD used by both admins and tenants. An example of the set command is given below:

```
1. az login
2. az login -u <username> -p <password>
3. read -sp "Azure password: " AZ_PASS && echo
&& az login -u <username> -p $AZ_PASS
4. $AzCred = Get-Credential -UserName <username>
5. az login -u $AzCred.UserName -p
$AzCred.GetNetworkCredential().Password
6. az login --service-principal -u <app-id> -p
<password-or-cert> --tenant <tenant>
7. $AzCred = Get-Credential -UserName <app-id>
8. az login --service-principal -u $AzCred.UserName
-p $AzCred.GetNetworkCredential().Password
--tenant <tenant>
```

Azure AD supports a wide variety of operating systems. Each of the operating systems uses different commands for different services (e.g., Virtual machines, resources, federated applications) offered.

As part of CoZure, we have developed a web scraping tool to gather different commands using different Azure AD keywords (e.g., login, storage, user-managed identity) are used to collect the

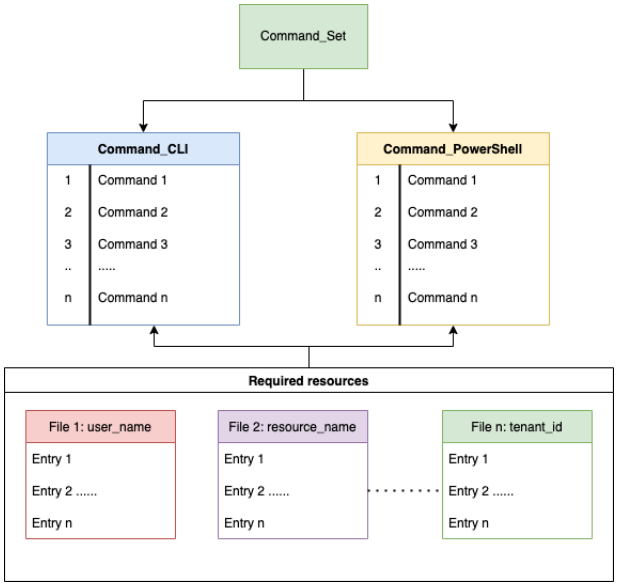

Figure 3: Command set

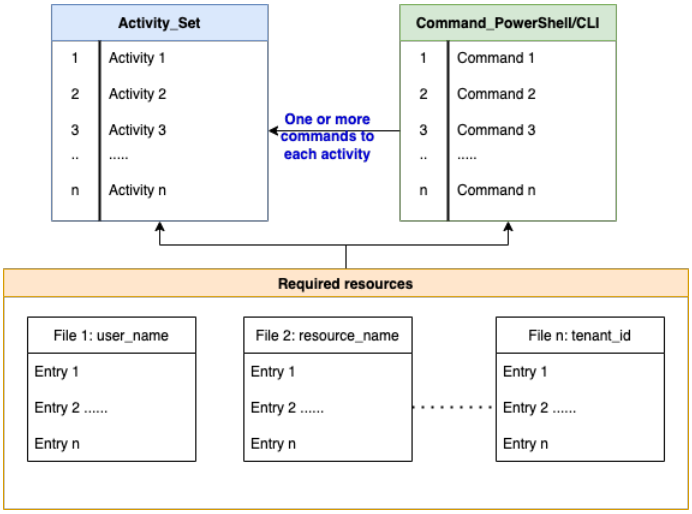

Figure 4: Activity set

command used in the Azure AD environment from Microsoft Office websites, forums describing Azure AD vulnerabilities and Azure AD support pages (e.g., https://docs.microsoft.com/en-us/cli/azure/authenticate-azure-cli). This tool first requests for the keywords and then requests the selection of URL(s), both keyword and URL can be single or multiple entries, or the source can be selected from a CSV file.

PowerShell and Command Line Interface (CLI) are the two major platforms used in the Azure AD environment. For this reason, we have created two different subsets of commands, one for CLI and one for PowerShell (ref to Fig. 3). For example, the different commands for adding a member to the Azure administrative unit in PowerShell and CLI are stored as shown below:

1. PowerShell environment:

```
Add-AzureADMSAdminUnitMember -Id <String> -RefObjectId
<String> [-InformationAction <ActionPreference>]
[-InformationVariable <String>][<CommonParameters>]
```

2. CLI environment:

```
$adminUnitObj = Get-AzureADMSAdminUnit -Filter
<displayname>$userObj = Get-AzureADUser-Filter
<UserPrincipalName> Add-AzureADMSAdminUnitMember -Id
$adminUnitObj.Id-RefObjectId $userObj.ObjectId
```

Note that some commands require input parameters (e.g., username, tenant_id) to be selected manually and may contain keywords such as Add, Get, and Change. The required information is usually displayed within specific characters (e.g., <>, “, []). We created a separate CSV file for our system for each required information (e.g., password, app-id). When a command requires any specific information, our system can take the information from the particular CSV file. If there are multiple entries, the tool takes the first entry first and checks if the command is successful or not; if the command is unsuccessful, the tool iterates through the remaining entries. At the moment, our command set contains 2230 commands for both CLI and Powershell.

3.2.2 Building the Set of Activities for AAD . The second set of entities in our model is the set of activities. An activity can contain one or more selected commands as shown in Figure 4. In our work, the set of activities is further subdivided into four different subsets of environments for CLI, PowerShell, Linux (Kali), and Windows (10 and 11). Creating new users, updating passwords, changing passwords, adding an RBAC role, creating an SSH key, and others are examples of the activities.

Each activity is linked with one or more commands from the command list. The following example shows an example activity where the admin creates a new user and a new password and forces the user to change the password at first login. This activity contains three different types of commands:

```
az ad user create --display-name --password--user-
principal-name [--force-change-password-next-login
{false, true}] [--immutable-id][--mail-nickname]
```

3.2.3 Building the Set of Security Attributes for AAD . The third critical entity in our model is the set of Security Attributes. Azure AD is a cloud-based identity and access management service that provides secure access to various applications and services in the Azure cloud environment. AAD has a number of security attributes that help to protect user identities, secure access, and manage permissions.

Each security attribute or task is linked with one or more commands or activities, or both. For example, adding a secret key to the service principle for a user (SPSKeyAD) is a security attribute. This attribute allows the user to log in to a service principle using the key. This attribute contains commands (e.g., password, AppName) and activities (e.g., GenSecret, AddSecToApp). The final command for this security attribute (SPSKeyAD) is

```
Add-SPSKeyAD -AppName [Application] -Password [secret]
```

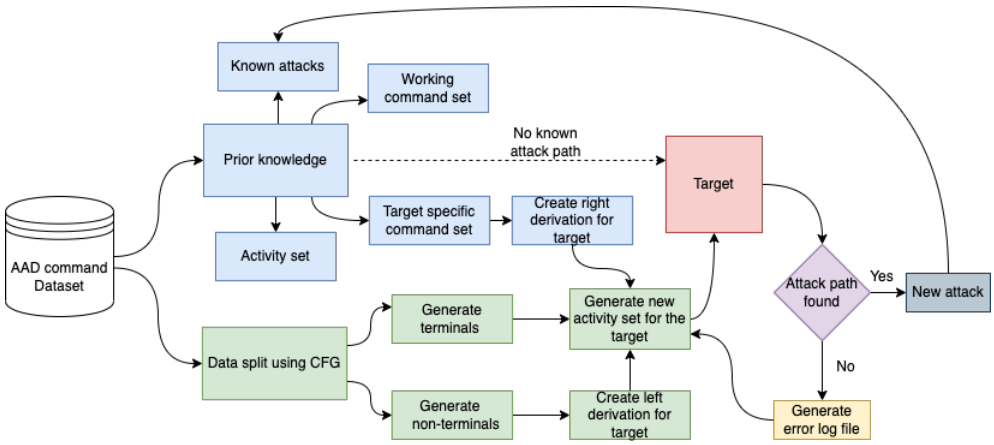

Figure 5: CFG model flow diagram for Azure AD

In our current dataset, we have 73 security attributes. The security attributes were created as per the guideline provided by Microsoft Azur AD documentation site [20].

3.2.4 Building the Set of Targets for AAD . The fourth and last set is called the set of known targets. Targets in an Azure AD environment are often assets and/or security principals. Security principals are Active Directory objects that can manage any object to which permissions are assigned to the principals.

Azure AD provides several services (e.g., storage, database management, federated application) and contains different components (e.g., virtual machines, resources). In our model, the target set contains all the services provided by Azure AD and components used by Azure AD. We created a list of 42 services used in Azure AD and added these to the target list. The number of targets can increase or decrease according to the number of services used in each organization. Each target is initially linked with the relevant commands, activities, and attributes used for this service or component.

The stored sets of activities can execute commands to get access to different resources, Azure AD, and third-party services. Some of the commands can start different Azure AD services (e.g., Virtual Machines) that may incur running costs. In our current implementation, all four sets of commands, activities, attributes, and targets are designed for the manual selection of targets and attacks to adhere to the university's network usage policy and avoid any unnecessary cost. users with permission to run vulnerability tools on the network can modify the code to run the tool automatically. The algorithm to make the code run automatically with the user selection option is given in GitHub.

3.3 CFG Model for Azure AD

The next step is to build a CFG model for the command, activity, attribute, and target sets built in the previous sections. Our CFG model contains the following key elements:

- The *alphabet* is the set of all the symbols in the input language. We have created two sets of alphabets: one for the Command line interface (CLI) and the other for the Power Shell environment. Each alphabet is a valid command (e.g., az login) of an Azure AD environment.
- A *terminal* is a single alphabet symbol. In this case, it is any command in our set of commands.
- A *nonterminal* is a symbol outside the alphabet whose expansion is defined in the grammar using rules for expansion. Nonterminal symbols assist in generating a sentence but not a part of the sentence. For example, in Azure AD, *ListManageIdentity* is a nonterminal activity, where *ListManageIdentity* can be expanded to *azlogin* and *Get-AzureManagedIdentity*.
- A *production rule* is a finite sequence of terms (terminals and nonterminals) that describe an expansion of a given terminal. For example, the CFG production rule $A \longrightarrow aAbB$ can be used to represent the Azure AD rule *KeyVaultAccess* → *az login StorageLogin openkeyvaultfile KeyVaultPlaintext*
- A *definition* is a set of rules that describe the expansion of a given nonterminal. Our model uses the list of definitions (e.g., MoveWin2Lin, PortForward, RemoteScriptSSH) stored in its database to explore new lateral movements in Azure AD.

Context-free grammar composes of a set of nonterminals and corresponding definitions that define the structure of the nonterminals. Typically, a CFG can be modeled using parse trees. The tree nodes represent the symbols, and the edges describe the use of production rules. The tree leaves are the results (terminal symbols) that make up the string that the grammar generates with that particular sequence of symbols and production rules. A *derivation tree or parse tree* can be collapsed into its string equivalent. Such a string can be parsed again by the nonterminal at the root node of the derivation tree such that at least one of the resulting derivation trees would be the same as the one we started with. We used left derivation and right derivation trees for our model, as explained below.

3.3.1 Derivation Trees. A left derivation tree is a tree structure that represents the process of deriving a given string from the start symbol of a context-free grammar by applying a sequence of leftmost derivations. Each node in the tree corresponds to a symbol

in the grammar, and each edge corresponds to a production that is used to derive the symbol in the parent node from the symbol(s) in the child node(s). The leftmost derivation of a string is a derivation in which the leftmost non-terminal symbol in the current string is always chosen for expansion. This means that at each step of the derivation, the leftmost non-terminal symbol is replaced by the right-hand side of a production that has the non-terminal symbol on the left-hand side. To construct a left derivation tree for a given string in a context-free grammar, we start with the root node of the tree, which corresponds to the start symbol of the grammar. We then apply leftmost derivations to the start symbol until we derive the input string. At each step of the derivation, we create a new node in the tree for the symbol that is being derived and connect it to the parent node(s) that correspond to the symbols that were used to derive the current symbol. For example, consider the following context-free grammar: $S \rightarrow AB, A \rightarrow a, B \rightarrow bB|cB|\ \epsilon$

Let the representation for this grammar rule be: S = "*az login*", A="Azure Resource A", B="KeyValut B" in Resource A, a=login to azure, b= connection to an azure resource, c=access to keyvault file. If we want the following sequence of commands to access vault file B in resource A:

```
a= az login -u <username> -p <password>
b= az login --identity --username <resource_id>
c= az keyvault key show [--hsm-name] | --vault-name,
[--id]|[--name], [--version]
NULL = No action.
```

Suppose we want to construct a left derivation tree for the string "abc". As shown in Figure 6(a), the leftmost derivation of this string is as follows:

$S \rightarrow AB, A \rightarrow aB, B \rightarrow bB, B \rightarrow bB, B \rightarrow \epsilon$

The corresponding left derivation tree for this string is shown in Figure 6. In this tree, the root node corresponds to the start symbol S, and has a child node for the non-terminal symbol A. The node for A has a child node for the terminal symbol "a", and the node for B has three child nodes for the terminal symbols "b", "c", and ϵ, respectively. The tree represents the leftmost derivation of the string "abc" in the given grammar. The Python code for generating the left derivation tree is given in GitHub.

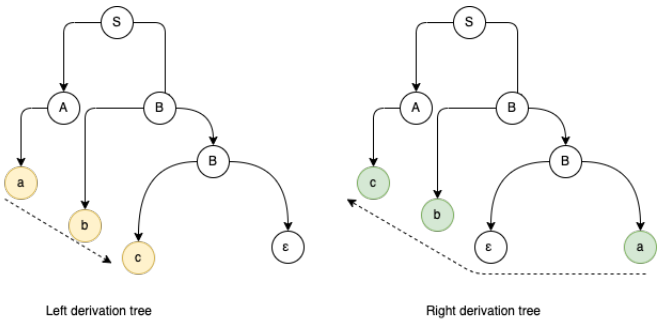

Figure 6: Left and Right derivation trees

Similarly, a right derivation is a derivation in which the rightmost non-terminal symbol in the current string is always chosen for expansion. This means that at each step of the derivation, the rightmost non-terminal symbol is replaced by the right-hand side of a production that has the non-terminal symbol on the left-hand side. To construct the right derivation tree for a given string in context-free grammar, we start with the root node of the tree, which corresponds to the start symbol of the grammar. We then apply the rightmost derivations to the start symbol until we derive the input string. At each step of the derivation, we create a new node in the tree for the symbol that is being derived and connect it to the parent node(s) that correspond to the symbols that were used to derive the current symbol.

3.3.2 Use of Left and Right Derivation . One of the most significant processes of our model involves using the left derivation, right derivation, and a combination of left and right derivation. When a user selects the option to explore vulnerabilities, our system initiates testing with known attacks. The sequence of commands for these known attacks is pre-stored in our database. Our model executes the commands stored in the system, following the left derivation. This means it systematically proceeds from the first (left-most) step or element, moving towards the last (right-most) when dealing with predetermined targets and activities. This approach ensures a systematic and orderly execution of commands. If all commands of a known attack are successfully executed, our model sends a notification to inform that the system is vulnerable to this known attack. If a specific command is not executable (restricted or protected by the Azure AD environment), the model interprets the task as secure, acknowledging the system's defensive capabilities. The model then stores this information in the parse table, which contains the start position, the last successfully executed command, the next required command, and the final target.

When the system successfully defends itself against a known task, the principal function of our model initiates the exploration of alternative paths to reach the target. In this case, the system utilises information about all successfully executed commands used in the left derivation from the parse table. Suppose we denote the first executed command as 1, the last successfully executed command as m, and the final command required to reach the target (execute the attack successfully) as n. Our model identifies all possible $n-1$ commands that can be used to reach n using the right derivation. It's important to note that the right derivation primarily uses the backward pointers from Earley's table. If multiple paths $(n-1)$ are available to reach n, the system checks if there are any set of activities available to reach $n-1$ or n from m or 1 to $m-1$ to bypass $m+1$, given the system protects that m to $m+1$. If no alternate path is available, the system checks all possible options from n to $m+1$ using a combination of left and right derivation. As shown in Figure (7), for Target 3, the model employs nonterminal commands such as 1, 5, and 19 before arriving at 3. If the path from 19 to 3 is obstructed, our model diligently seeks an alternate route to Target 3.

During the training phase, our model utilises the Earley parsing technique, storing all the backward pointers for each parsing state. If we consider 3 as the n^{th} task for Target 3, then the preceding command, 19, is the $(n-1)^{th}$ task. With the current $(n-1)^{th}$ to n^{th} task pathway obstructed, our model investigates potential alternative routes to Target 3. As outlined in Figure (7), paths through 2, 19, and 27 lead to Target 3. Given that the path from 19 to 3 is

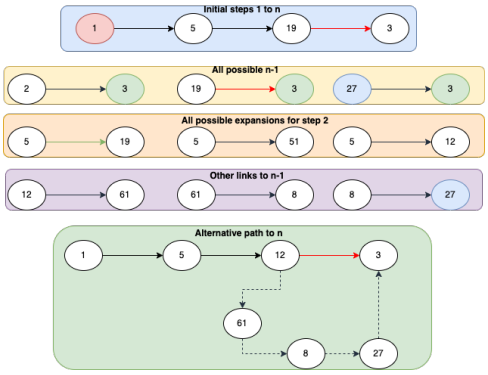

Figure 7: Use of Left and Right Derivation

blocked, our model investigates any feasible pathways to 2 or 27 from Steps 5 or 19. Despite the lack of a direct route, our model systematically expands Task 2 (5) to identify any possible pathway to 2 or 27.

The combination of left and right derivation techniques, along with the backtracking options facilitated by Earley's chart, contributes to the superior efficiency and effectiveness of our model over other tools reliant on predefined activity sets for the exploration of known tasks in the observed environment.

3.3.3 Training the CFG model . To train our CFG model, we need the sets of commands, activities, attributes, and targets as explained in Section 3.1. In addition to these sets, we need to feed the model with how these entities are combined in a successful attack. To do this, we collect a set of known lateral movements, and for each of these lateral movements, we build an ordered list of sequences of commands used in the attacks and their targets. So far, we have trained CoZure with 17 recent lateral movements on AAD [6, 10, 11, 15–17, 28, 29].

To start the training phase of our model, we first execute the known lateral movements, working commands, and set of activities relevant to the specific target. During this training phase, our model stores the set of activities and targets specific commands or activities.

A list of the working commands is saved as the terminals, and a list of the activities is saved as non-terminals. The commands and activities used in the known lateral movements are saved as the known production rule. We generate the right derivations for the targets from these known production rules.

After training the CFG model with known lateral movements, we can start expanding the model to explore new lateral movements. Once our model starts executing a set of commands or activities, when any new attributes or service is added, or we want to test by replacing any existing activity with a different activity or set of activities, we use CFG, left and right derivation [13] to find the closest match for the replacement. Note here that the execution requires access to a real AD environment. If the execution is successful and the activity is not listed in the known activity set, our model saves the sequence of the commands as a new activity. The tool then checks this new activity to see if this activity is part of any known attack (already stored in the database). If the activity is not a new lateral movement, the activity is added to the known activities list.

If the execution of the command is unsuccessful, CLI or Powershell will generate an error code. Cozure saves the error code details. In the general case of the Azure AD environment, the error codes can be manually checked. A step-by-step walkthrough of the training of CFG on the attack in [16] is given in Appendix A. CoZure learns a number of grammar rules, terminals, and non-terminals from this attack. Some of the grammar rules, terminals, and non-terminals are given below:

```
Terminals
1. az login --identity
2. az keyvault list
A. Non-Terminal (NewUserAssignIdentity)
1. az identity create -g Demo -n myUserAssignedIdentity
--location westEurope
2. az vm create --name WindowsVM --resource-group Demo
--image <> -- location westeurope -- admin-username <> --
admin-password <>  --assign-identity/subscriptions/
<myUserAssignedIdentity>
3. az role assignment create --assignee <> --role
Contributor --scope /subscriptions/<WindowsVM>
```

Using grammatical models and production rule substitution, we can check whether any alternative path or commands can be found - we will explain this application of CFG to lateral movements discovery in the next section.

3.4 CFG Algorithm for Finding New lateral Movements

The complete algorithm for finding new lateral movements in Azure AD using CFG is shown in Algorithm 1. To find the new lateral movements, we convert the sequence of actions in an existing lateral movement into "strings" that contain all the sequences of commands in that attack. Recall that in CFGs, a string is a sequence of terminal symbols that can be generated by a CFG starting from the start symbol. The start symbol is a non-terminal symbol that represents the initial category or type of the string. To generate a string in a CFG, we start with the start symbol and use production rules to rewrite it into a sequence of other symbols, which can be either non-terminal symbols or terminal symbols. We keep applying the production rules until we have a string consisting entirely of terminal symbols.

We use two different parsing algorithms named CYK [12][7] parsing algorithm (bottom-up approach) and Earley's [3](top-down approach) parsing algorithm to parse the strings. Using bottom-up and top-down approaches gives us a better opportunity to understand the string and options to use both left-to-right and right-to-left derivation methods. These options are essential to predict the best possible activities while exploring new lateral movements. The remaining part of this section describes how parsing works for strings and how we use it to parse the attack activities.

Assuming that we have the following grammar rules (learnt from the steps described in Algorithm 1) with starting symbol S, terminals with small letters, and nonterminals with capital letters.

```
S -> NP VP, PP -> P NP, VP -> V NP | VP PP
NP-> NP PP | Det N, P -> resourceGroup
```

Algorithm 1: Algorithm for finding new lateral movements

```
Input: The set of commands, activities, attributes, targets
       and known attacks
Output: List of Azure commands, set of related commands,
        lateral movement status
Step 1: Create the initial grammar rule
start_symbol = "az"
terminals, non_terminals =
 generate_terminals_and_non_terminals(commands)
save_grammar_to_file(start_symbol, terminals,
 non_terminals)
Step 2: Execute known movements
for attack_command in attack_commands do
    execute_command(attack_command)
end
Step 3: Building dataset for Azure AD
azure_ad_commands = create_azure_ad_commands()
label_commands(azure_ad_commands)
save_commands_to_file(azure_ad_commands)
Step 4: CFG grammar rule creation
for command in azure_ad_commands do
    execute_command(command)
    save_related_commands()
end
use_parsing_algorithm()
use_derivation_algorithm()
use_graph_algorithm()
use_table_algorithm()
Step 5: Finding new lateral movements
target = select_target_to_test()
if target in known_targets then
    execute_known_commands(target)
end
else
    related_commands = find_related_commands(target)
    for command in related_commands do
        if target in command then
            execute_command(command)
            if lateral_movement_found() then
                stop_searching()
            end
            else
                continue_searching()
            end
        end
    end
end
```

```
NP -> az | login |batch |storage| pool
V -> deployment | group | writes
N -> name | tag, Det -> create | location
```

When we use the CYK parsing algorithm to test the following three different sentences for grammar $\mathcal{G}$ and language $\mathcal{L}$:

1. *az group create name resourceGroup location tag* (**Accepted** sentence in $\mathcal{L}$)

2. *az group create name resourceGroup tag* (**Not accepted** sentence in $\mathcal{L}$)

3. *az group createss name resourceGroup location tag* (contains words **not in** $\mathcal{G}$)

For Sentence 1, the sub-strings for length one are NP → az, V → group, Det → create, N → name, P → resourceGroup, Det → location, and N → tag. The applied rules for creating a parse table for Sentence 1 are given below. The number given within bracket ([ ]) presents the position and length of the parsing table (refer to [12] for details about the CYK parsing table).

```
Applied Rules:
NP[2,3] --> Det[1,3] N[1,4], NP[2,6] --> Det[1,6] N[1,7]
VP[3,2] --> V[1,2] NP[2,3], PP[3,5] --> P[1,5] NP[2,6]
[4,1] --> NP[1,1] VP[3,2], NP[5,3] --> NP[2,3] PP[3,5]
VP[6,2] --> V[1,2] NP[5,3], VP[6,2] --> VP[3,2] PP[3,5]
S[7,1] --> NP[1,1] VP[6,2], S[7,1] --> NP[1,1] VP[6,2]
----------------------------------------
The sentence IS accepted in the language
Number of possible trees: 2

The applied rules we can get for Sentence 2 are:
NP[2,3] --> Det[1,3] N[1,4]
VP[3,2] --> V[1,2] NP[2,3], S[4,1] --> NP[1,1] VP[3,2]
----------------------------------------
The sentence IS NOT accepted in the language
----------------------------------------
```

For Sentence 2, if we add a determiner (Det) before the last terminal, the sentence will be accepted as the correct sentence. For Sentence 3, no applied rule information or parse tree information is received.

In this case, if we replace the terminal (*createss*) with a correct known (using prior knowledge or a list of terminals), we can also get a proper sentence.

Note here that the key task in the algorithm is to substitute the possible entries to generate the correct string. In lateral movement path exploration, our algorithm uses different commands or activities as grammar rules and tries to find any combination that leads to a successful pivot.

4 COZURE APPLICATIONS

4.1 Cozure: AAD lateral movement discovery tool

We built a software tool, called *CoZure*, for our model in Python using libraries from Early parser [3]. CoZure can be executed from a local (on-premise device) or Azure AD environment. To run CoZure from an on-premise device, the user needs to provide the Azure AD credential at the beginning to authenticate. CoZure has three options:

- Option 1 is to select any know attacks, and CoZure follows the predefined set of instructions from a set of known attacks and reproduces the attack. This helps analyze the attacker's steps to draw graphs and log files for further investigation.

- Option 2 is to select a specific target from the target set. CoZure selects the required attributes, activities, and commands used for the target and randomly uses different combinations to explore new lateral movements.
- Option 3 is to select any target and add different attributes or activities from the listed set or the user-defined new set to investigate whether a new lateral movement is found when a new resource or attribute is added.

Each time we run CoZure either with an existing attack or with Option 3 (try to explore a new lateral movement), CoZure will find if there is any new successful activity and if the new activity is identified, it will add the activity to the existing dataset. This improves the prior knowledge of our model and helps CoZure find new lateral movements with improved prior knowledge.

4.2 Reproducing known attacks

As mentioned in Section 4.1, Cozure has the option to reproduce known attacks (Option 1). Running the CFG model with known attacks could also be viewed as automating the detection of *n*-day vulnerabilities in an Azure AD system. These vulnerabilities and attacks are known, but the targeted AAD system is not yet patched for them. Our tool can perform these attacks by itself, given the corresponding grammar rules that the model learns from parsing a previous successful attack.

To reproduce known attacks, the existing attack details that are collected during the data collection and data processing steps are fed into our model. We go through the details of two attacks in the rest of this section.

4.2.1 Password brute force attack . We show here how CoZure reproduces the password brute force attack in [6]. To use the tool, a user (an expert hacker) needs to create and insert the grammar rules for this attack into our model. We start by defining the non-terminal symbols:

```
S=az, a=username, b=password,
c=hashkeys, d=tenant, e=select
```

We then follow by defining the terminals

```
A=Login, B=Create, C=set, D=role, E=-U
```

assignment and set the following production conditions:

```
S -> aAS|aSS|E, A -> SbA|ba.
```

Given these initial sets, we can generate strings such as *aabaa* by following the rules we define for the grammar. For the password (P) command

```
az login -u <username> -p <password>
```

the rule will be $P \rightarrow SAEab$, where terminal symbol a represents users and b represents passwords.

If $H = storagelogin$ and the command is

```
az storage (s) login username,
```

the rule will be $H \rightarrow PSsAa$.

Note that the rules are manually encoded and inserted in CoZure. Given these inputs, CoZure then adds new commands as terminals and activities as non-terminals and assigns the grammar rules automatically. To reproduce the brute-force password attack in [6], we input the command:

```
az login -u <username> -p <password>
```

CoZure selects the first password saved in the password file and tries this password with all of the listed usernames. CoZure is designed in such a way that it can search for the relevant CSV file if the portion of the command has angular brackets (<>). CoZure then selects other passwords one by one and multiple counters to go through all listed users and passwords.

Given the CFG model specifications above, the tool will start with validating the user first (P) and then perform the activities for storage login, eventually reproducing the attack in [6]. The complete data and code for this attack are provided in GitHub

4.2.2 Lateral movement attack . We explain here how our tool reproduces the sophisticated attack in [16] that we describe in Section 2 using the CFG model.

We first gathered the following information from [16]:

- List of commands: We gathered a total of 41 commands used in this attack. These commands were added to the list of command files.
- List of activities: A total of 17 activities, such as creating virtual machines, adding new users, and Azure storage login, were added to the list of activities files.
- This attack also has eight Azure AD security attributes such as tenant IDs, user-managed identities, and Microsoft Azure extension.
- The known target list was updated with the Azure storage, Virtual Machine, KeyVault file, Network Security Group, and open ports.
- Three different attacks with the sequence details were updated from this attack details. The first attack is key vault access, the second attack is moving from Linux to Windows machine, and the third attack is moving from Linux to Linux machine.

Once we encoded these symbols and the corresponding rules into our model, we ran our tool with Option 1 and selected the attack name. The tool successfully regenerated the attacks. Detailed code and data for this attack are provided in GitHub.

4.3 Explore new lateral movements

We have used CoZure to extend lateral movements in [16] to find new ways that an attacker could use to move laterally to different VMs and different services by exploiting access delegations in Azure AD. We tested and verified these attacks on an Azure environment similar to the one in [16]. We used only three virtual machines and two resources and did not use any third-party applications to stay within the guideline provided by Microsoft. If the CoZure is implemented in a larger network, there will be more possible attack paths.

To enable the model to find new lateral movements, we first used the possible grammar substitution rules to identify three major points where an attacker can take other possible movements than the ones detailed in [16]. Each activity point is considered non-terminal in CFG. As per Earley's algorithm, each state expands multiple predictions available for the non-terminal. This prediction allows the model to check all the different possible options. Every option may lead to a new possible attack vector. The possible attack vectors are shown in the red boxes in Figure 8. These three attack vectors could be instantiated by the two possible new exploits :

(1) Using RDP tunnelling through an open port to bypass RDP restrictions,
(2) Moving from a compromised Linux to a target Linux account.

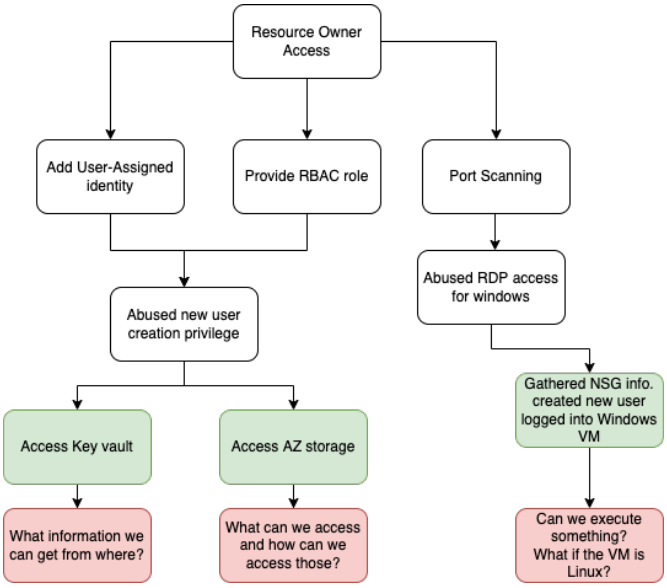

Figure 8: Possible new attack vectors

4.3.1 New RDP tunneling options . Our tool is pre-trained with activities to perform the attack described in [16] as explained in Section 4.2.2.

To force the model to find new lateral movements, we deleted the NSG modification privilege from the user and stopped RDP incoming permission on the target machine. We then instructed (by selecting Option 3) the tool to perform other activities in the available activities. More specifically, the tool tried a port scan to find open ports for remote connections, file transfer, and file executions. To do this, we provide a list of candidate ports saved in a CSV file named open_ports.CSV.

Note that the activity set contains 281 different options in our current version. The tool could select any random activity from this list. The critical thing here is that it would take considerable time to test all possible combinations of these activities. CFG with left and right derivation helps guide the activities, as explained below. We explore the performance aspect of our tool in Section 4.4.

As shown in Figure 9, the field labeled with 1 is the target machine, the field marked with 2 is the RDP connection, and the author in [16] used the field marked with the number 4 (modify NSG rule to open RDP port) following number 9, 12, 6, and 5 activity to reach number 4. When Option 4 is closed from number 5, our tool found other options (number 15 and number 16) available to reach number 4. Number 15 is to open the RDP port from the target machine; we do not have any credentials to log in to the target machine; we used numbers 18, 17, and 16 to activate port forwarding. Using CFG for possible replacement, our tool also was able to find the keyword for RDP or port 3389 in the port forwarding activity. The tool has chosen this activity to try first due to the closest match. Similarly,

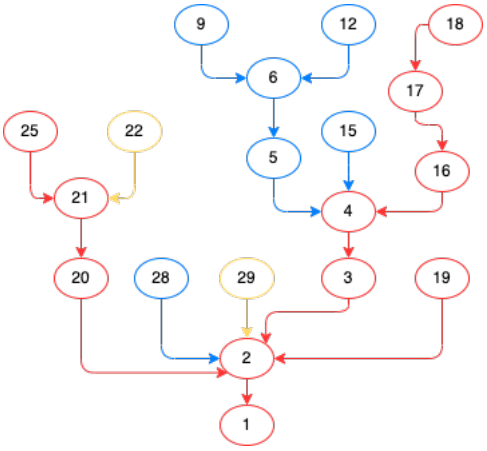

Figure 9: Use of CFG and right derivation

when the SSH connection was restricted, and we wanted to check if any alternate options were available to making the connection, using CFG and right derivation, our tool found activities that are more likely to be used as an alternative.

To verify the new lateral movements found by our tool, we ran the attacks on our Azure test environment. When our model executed some commands, one of the commands was for the port scan. Port 22 was found open for TCP connections (it is open in most systems by default). In this case, the model tries to use the port forwarding option as one of the other possible vulnerabilities related to "port" to allow RDP through SSH, using a command listed as port_forwarding under the "set of activity" file:

```
plink.exe test@<target_ip> -pw TestP@ssWord1!
-P 22 -2 -4 -T -N -C -R
0.0.0.0:12345:127.0.0.1:3389
```

In this command, port 22 (version 2) is used for making the connection to port 12345, which will be forwarded to port 3389. Once the SSH connection is established, the attacker can send an RDP connection request to 192.168.0.154:12345. The target Windows machine will consider this as an SSH connection but not an RDP connection.

Azure AD resource-managed identity service allowed us to make an SSH connection using port 22. We activated the alert generation in the Windows machine for the RDP connection, but it was unable to detect the RDP connection as the connection is showing source and destination ports as 22 and 12345, not 3389.

As per our best knowledge, there is no known attack using port forwarding and tunneling. Even though the author in [16] showed the attacker can use an RDP connection, the author mainly used the NSG rule modification. CoZure found that port forwarding is also possible. This port tunneling method also usages the stored keys for Azure AD, which makes the attacker model easier to perform the attack. Detailed code and data for this attack are provided in GitHub.

4.3.2 Finding new vulnerable ports . The attacks in [16] focus on connecting to Linux Machine using an SSH password-less connection and showed that the attacker could make an SSH connection

using only SSH keys and does not require any password for SSH. The target account is in the same resource group in Azure AD and locally in the target Linux VM. The command attacker can use is:

```
ssh targetaccount@192.168.0.154
```

This connection was possible as the target VMs was accepting password-less SSH connection. The SSH connection might not always accept password-less connections but won't work if password authentication is disabled.

We used CoZure to find new ways of making connections rather than SSH (as used in [16]) when the target machine restricts direct SSH password-less connections or connections with root privilege. For this, we let the tool try other options (automatically) when connecting to a remote machine in Azure. CoZure checked different possible options to see if it could utilize other ports (rather than 22) to drop infected payloads, gain a foothold, and remote code execution.

CoZure found several ways of remote connections other than SSH in our test environment. More specifically, it found that FTP/SFTP is possible with or without SSH keys. This allowed us to use other vulnerable ports with lower privilege levels (when the root connection is restricted). Again, the new lateral movement is discovered using CFG, left and right derivation. In particular, our tool tried the activities that contained the word "port" as the second last activity and found the relevant commands related to "port" and successfully executed the code. We manually verified this in our test Azure environment and found that we could do FTP/SFTP for lateral movement. Detailed code and data for this attack are provided in GitHub.

4.4 Comparison with other tools

We cross-compare our solution against existing vulnerability and lateral movement path-finding tools in this section. In particular, we compare our solution against BloodHound, Restlr, OneFuzz, and PowerZure. Note that BloodHound, Restlr, OneFuzz, and PowerZure are designed to only check if a given attack would be successful but could not generate new attacks themselves. For comparison, we assume that these tools use a random attack activity in our set of activities to generate new potential attacks and then check if these random variations constitute a genuine attack.

All of these vulnerability discovery and lateral movement analysis tools (including our tool) require running Linux virtual machine and one Windows virtual machine on Azure AD to perform the tests. Running these tests incur costs both time and financially. For example, the hourly running cost for these two VMs, resources, and storage on Azure AD is $33. It is therefore important that we limit the time that the different solutions use to find vulnerabilities. Recall that in our current activity set, there are 281 items. Randomizing through them could be very costly.

We compare CoZure against these tools (BloodHound, Restlr, OneFuzz, PowerZure) with random tries on four different scenarios.

(1) **Scenario 1: finding open ports in Azure AD** The first test aims to find the number of open ports in the Azure AD virtual machine.
(2) **Scenario 2: finding possible RDP connections in Azure AD** The second test aims to determine the possible RDP connection where the RDP port was open

Table 1: Comparison of using random method and using CFG

Test	Method	No. of tries	Time (sec)	CPU usage (%)	Cost of running (AUD)
1	Random	82	172.96	6	1.59
	With CFG	38	63.07	5	0.58
2	Random	197	407.43	17	3.73
	With CFG	112	176.39	13	1.62
3	Random	1376	2927.78	22	26.84
	With CFG	516	923.77	18	8.47
4	Random	2189	3458.28	36	31.70
	With CFG	924	1967.95	24	18.04

(3) **Scenario 3: finding port forwarding options in Azure AD** In the third test, we closed the RDP port and ran the test for exploring port forwarding.
(4) **Scenario 4: finding options to perform remote code execution in Azure AD** The final test is to explore the option to perform remote code execution.

Scenario 1 represents the password brute force attack. As the attack was generated from outside the Azure AD network and there was no misconfiguration in Azure AD, the graph generated by BloodHound did not create any alert for the attack, and Restlr and OneFuzz also did not generate any alert. PowerZure has the option to perform this type of external attack. Our tool can check internal and external threats even though there is no misconfiguration issue.

In Scenario 2 RDP port is open, and the NSG rule has the option for modification. All of the tools, including ours, can detect the vulnerability as the port scan and the access privilege of the contributor show the possible lateral movement.

When the RDP port is closed, and NSG rule modification is restricted in Scenario 3, PowerZure does not generate any alert for the RDP connection, BloodHound, OneFuzz, and Restlr showed the other open port details but unable to identify the RDP connection still be possible. Our model can identify the possibility of an RDP connection by adding rules (for port forwarding) that allow the user to get an RDP connection.

For Scenario 4, we consider the vulnerable code execution on the virtual machine, no tools rather than our tool found the option to explore the remote code execution option. PowerZure is able to perform such a task but it needs manual intervention or instruction from the human operator to perform such task.

As shown in Table 1, we can see that the number of tries across all four scenarios is significantly lower when we use CFG. Due to the fewer attempts, the execution time and running cost are also lower. The reason behind the fewer tries is that, with CFG, CoZure uses the left and right derivation method to find the next activity and does not use some activities such as "ListManagedIdentities", "RunAsAccounts", "SSHKeys", "PortForwarding", "PrivilegedIdentityManagementAssignments" that are not required for the task.

5 CONCLUSION

We developed CoZure to help researchers find new lateral movements in an Azure AD. The parsing and derivation techniques show the possible expansion of non-terminal and different options

to reach the target (terminals). Left and right derivation also allow the analysts to understand and investigate possible alternate paths to reach a source from a destination. Exploring new lateral movements in any environment (e.g., Azure AD) needs strong prior knowledge about the environment, and analysts need to spend a significant amount of time using the prior knowledge to find out new attack methods. Our model is capable of improving its prior knowledge by analyzing existing attacks, current penetration testing tools, and vulnerability exposure tools. Earley's parsing algorithm and context-free grammar enhanced the capability of our tool to detect existing misconfiguration issues and explore new lateral movements with reduced resource uses and less human intervention.

In future works, investigation on probabilistic context-free grammar and parsing algorithms can be used to generate the sub-trees for activities, analyze the log file for error messages in every state, and predict with probabilistic values will increase the chances of exposing new attack edges. These will also allow us to improve our tool in such a way that it can dynamically learn from the output in each state to improve efficiency.

Vulnerability Disclosure Process (VDP): *User must adhere strictly to the penetration rules of engagement as stipulated by Microsoft. Our VDP includes three major elements: An internal VDP, An external VDP, and Reporting and communication channels, including key points of contact. Identification of vulnerabilities requires meticulous documentation and verification. The report identified vulnerabilities in the respective organization securely, ensuring the information was clear and fostering further collaboration to resolve the vulnerabilities. In disclosing vulnerabilities, align with the Australian Cyber Security Centre's guidelines, thereby granting the organization sufficient time to manage the vulnerabilities prior to their public disclosure.*

ACKNOWLEDGMENTS

Australian Research Council - National Intelligence and Security Discovery Research Grants (ARC-NISDRG grant number NI210100139)

REFERENCES

[1] Aleph. 2018. DOS vulnerability in Azure Active Directory Graph API. https://alephsecurity.com/vulns/aleph-2018003

[2] Vincenzo Arceri, Martina Olliaro, Agostino Cortesi, and Isabella Mastroeni. 2021. Completeness of string analysis for dynamic languages. *Information and Computation* 281 (2021), 104791.

[3] John Aycock and R Nigel Horspool. 2002. Practical earley parsing. *Comput. J.* 45, 6 (2002), 620–630.

[4] Afnan Binduf, Hanan Othman Alamoudi, Hanan Balahmar, Shatha Alshamrani, Haifa Al-Omar, and Naya Nagy. 2018. Active Directory and Related Aspects of Security. In *2018 21st Saudi Computer Society National Computer Conference (NCC)*. NCC, Saudi Arabia, 4474–4479. https://doi.org/10.1109/NCG.2018.8593188

[5] Microsoft Research Blog. 2020. *RESTler finds security and reliability bugs through automated fuzzing.* Microsoft. https://www.microsoft.com/en-us/research/blog/restler-finds-security-and-reliability-bugs-through-automated-fuzzing/ (Last accessed on: 30/06/2023).

[6] L Bošnjak, J Sreš, and Bosnjak Brumen. 2018. Brute-force and dictionary attack on hashed real-world passwords. In *2018 41st international convention on information and communication technology, electronics and microelectronics (mipro)*. IEEE, Opatija, Croatia, 1161–1166.

[7] J-C Chappelier and Martin Rajman. 1998. A generalized CYK algorithm for parsing stochastic CFG. In *Proc. of 1st Workshop on Tabulation in Parsing and Deduction (TAPD'98)*. INRIA, France, 133–137.

[8] Mao Chenyu and Guo Fan. 2016. Defending SQL injection attacks based-on intention-oriented detection. In *2016 11th International Conference on Computer Science & Education (ICCSE)*. IEEE, Xiamen, China, 939–944.

[9] Abdullahi Chowdhury, Gour Karmakar, Joarder Kamruzzaman, and Tapash Saha. 2018. Detecting intrusion in the traffic signals of an intelligent traffic system. In *Information and Communications Security: 20th International Conference, ICICS 2018*. Springer, Lille, France, 696–707.

[10] XM Cyber. 2021. *Privilege Escalation and Lateral Movement on Azure – Part 1.* XM Cyber. https://www.xmcyber.com/privilege-escalation-and-lateral-movement-on-azure-part-1/ (Last accessed on: 30/06/2023).

[11] Ryan Hausknecht. 2021. *Attacking Azure & Azure AD, Part II.* Specter Ops. https://posts.specterops.io/attacking-azure-azure-ad-part-ii-5f336f36697d (Last accessed on: 30/06/2023).

[12] Jane C Hill and Andrew Wayne. 1991. A CYK approach to parsing in parallel: a case study. *ACM SIGCSE Bulletin* 23, 1 (1991), 240–245.

[13] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[14] Alicex Johndunagan, Zheng, and Simon. 2009. Heat-Ray:Combating Identity Snowball Attacks Using Machinelearning, Combinatorial Optimization and Attack Graphs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09* (2009). ACM Press, Big Sky, Montana, USA, 305. https://doi.org/10.1145/1629575.1629605

[15] Stuart Kwan. 2021. *Introducing Azure AD custom security attributes.* Tech Community Microsoft. https://techcommunity.microsoft.com/t5/azure-active-directory-identity/introducing-azure-ad-custom-security-attributes/ba-p/2147068 (Last accessed on: 30/06/2023).

[16] m365guy. 2021. *Lateral Movement With Managed Identities Of Azure Virtual Machines.* m365internals. https://m365internals.com/2021/11/30/lateral-movement-with-managed-identities-of-azure-virtual-machines/ (Last accessed on: 30/06/2023).

[17] Sean Metcalf. 2021. *Active Directory Security.* Ad Security. https://adsecurity.org/?p=4277 (Last accessed on: 30/06/2023).

[18] Microsoft. 2015. MS15-096: Vulnerability in Active Directory service could allow denial of service: September 8, 2015. https://bit.ly/3e9WDq7

[19] Microsoft. 2020. *Project OneFuzz.* Microsoft. https://www.microsoft.com/en-us/research/project/project-onefuzz/ (Last accessed on: 30/06/2023).

[20] Microsoft. 2022. *What are managed identities for Azure resources?* Microsoft. https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview (Last accessed on: 30/06/2023).

[21] OWASP. 2022. *OWASP Top Ten.* OASP. https://owasp.org/www-project-top-ten/ (Last accessed on: 30/06/2023).

[22] Una-May O'Reilly, Jamal Toutouh, Marcos Pertierra, Daniel Prado Sanchez, Dennis Garcia, Anthony Erb Luogo, Jonathan Kelly, and Erik Hemberg. 2020. Adversarial genetic programming for cyber security: A rising application domain where GP matters. *Genetic Programming and Evolvable Machines* 21, 1 (2020), 219–250.

[23] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript web applications using SAFE 2.0. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, Buenos Aires, Argentina, 59–62.

[24] Andy Robbins, Rohan Vazarkar, and Will Schroeder. 2020. AzureHound. https://bloodhound.readthedocs.io/en/latest/data-collection/azurehound.html

[25] Andy Robbins, Rohan Vazarkar, and Will Schroeder. 2022. *BARK.* BloodHound. https://github.com/BloodHoundAD/BARK

[26] Andy Robbins, Rohan Vazarkar, and Will Schroeder. 2022. *BloodHound.* BloodHound. https://github.com/BloodHoundAD/BloodHound

[27] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science* 88, 2 (1991), 191–229.

[28] AX Sharma. 2021. *New Azure Active Directory password brute-forcing flaw has no fix.* ARS Technica. https://arstechnica.com/information-technology/2021/09/new-azure-active-directory-password-brute-forcing-flaw-has-no-fix/ (Last accessed on: 30/06/2023).

[29] Marius Solbakken. 2021. *Quick look at managing Azure AD Custom Security Attributes using Graph.* Marius Solbakken Blog. https://goodworkaround.com/2021/12/01/managing-azure-ad-custom-security-attributes-using-graph/#more-59126 (Last accessed on: 30/06/2023).

[30] Colin Tankard. 2012. Taking the management pain out of Active Directory. *Network Security* 2012, 4 (2012), 8–11. https://doi.org/10.1016/S1353-4858(12)70025-9

[31] Shannon Williams. 2021. Businesses under threat as attackers target Active Directory. https://itbrief.com.au/story/businesses-under-threat-as-attackers-target-active-directory. Accessed: 30/06/2023.

[32] XU Zhiwu, Kerong Ren, and Fu Song. 2019. Android malware family classification and characterization using CFG and DFG. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, Guilin, China, 49–56.

APPENDIX A

Walkthrough of CoZure on the lateral movement attack [16]. In this attack (explained in Section 2.2), the attacker has four targets, which we denote abstractly for brevity as $\{A, C, E, G\}$, four activities $\{B, D, F, H\}$, and 12 commands

$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$.

For illustration, we assume that each activity uses three commands (or a set of commands). To be successful, the attacker needs to successfully execute all the commands in an activity before they can go to the next target.

To parse this attack using CFG, we first define the set of grammar rules as :

$S \rightarrow A|C|E|G$
$G \rightarrow HH, H \rightarrow HH|[10, 11, 12]$
$C \rightarrow DD, B \rightarrow BB|S|[1, 2, 3]$
$E \rightarrow FF, D \rightarrow DD|S|[4, 5, 6]$
$A \rightarrow BB, F \rightarrow FF|S|[7, 8, 9]$
$Input$: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

Here S is the starting symbol, which means the attacker starts the first command here and will come back to this part to check if the target has been reached and the attack is completed or not.

The objectives of the attack are to create a new user with a user-managed identity, modify the RBAC role, and access Azure Storage; and the final target is KeyValut file access (in plain text). To achieve the targets, the attacker needs to use different activities such as creating a new user, assigning the user to a specific resource, modifying RBAC role privilege, and different commands, such as:

```
. az keyvault secret list --vault-name
mysecretkeyvault01
. az ad user create --display-name "New" --password "<P>"
. az role assignment create --assignee "NU" --role
"Contributor" --scope "<subscriptions>/<resourceGroups/>"
. az storage account list
. az keyvault secret list --vault-name mysecretkeyvault01
. az storage blob list --account-name mystorageaccount
--container-name mycontainer
.az keyvault secret show --<vault-name> --name mysecret
```

The attack is a sequence of commands from 1 to 12. Once the attacker successfully executes command 12, the attack is successful. If the attacker does not follow the grammar rule, the attack will not be successful.

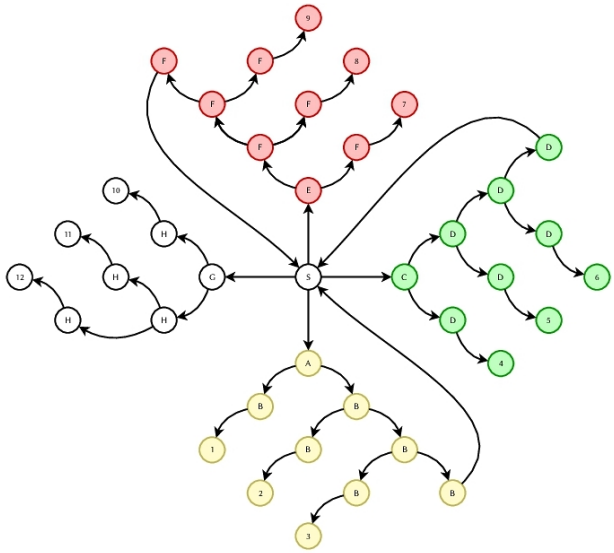

Figure 10: Parsing tree derived using Earley's parsing

The parsing using Earley's parsing algorithm generates the parsing table. Table 2 shows the first three and last state of this parsing. The header column represents the state. The process starts from the initial state (init) where it expands (annotated as prediction) all possible options (targets) from the start screen. The prime target of this state is to show the option for completing the first task (execute command 1). Once the path for executing task one is found, it goes to the next state (State 1) to scan the output of task 1 and set this as completed. Once the first three tasks (Task 1, 2, and 3) are completed, it also marks the first activity (B) and first target (A) as complete. This process goes on until the final command (Command 12) is successfully executed, and the parser sets the whole process as complete at State 12.

The parsing tree shown in Figure 10 is a graphical interpretation derived from the parsing table. We can see that this tree has different subtrees (set of tasks) as marked with different colours in Figure 10. These subtrees or activities are saved in our set of activity list for exploring new attacks.

Table 2: Earley table showing different states, scans, predictions, and completion steps

init	1	2	3	4	5	6	7	8	9	10	11	12
$_0$ .S Pred. $_0S \rightarrow .A$ $_0S \rightarrow .C$ $_0S \rightarrow .E$ $_0S \rightarrow .G$ $_0A \rightarrow .BB$ $_0B \rightarrow .S$ $_0B \rightarrow .1$	Scan $_0B \rightarrow 1.$ Comp. $_0A \rightarrow B.B$ Pred. $_1B \rightarrow .BB$ $_1B \rightarrow .2$	Scan $_1B \rightarrow 2.$ Comp. $_1B \rightarrow B.B$ Pred. $_2B \rightarrow .BB$ $_2B \rightarrow .3$	Scan $_2B \rightarrow 3.$ Comp. $_2B \rightarrow B.B$ Pred. $_3B \rightarrow .S$ $_3S \rightarrow .C$ $_3C \rightarrow .DD$ $_3D \rightarrow .4$	Scan $_3D \rightarrow 4.$ Comp. $_3C \rightarrow D.D$ Pred. $_4D \rightarrow .DD$ $_4D \rightarrow .5$	Scan $_4D \rightarrow 5.$ Comp. $_4D \rightarrow D.D$ Pred. $_5D \rightarrow D.D$ $_5D \rightarrow .DD$ $_5D \rightarrow .6$	Scan $_5D \rightarrow 6.$ Comp. $_5D \rightarrow D.D$ Pred. $_6D \rightarrow D.D$ $_6D \rightarrow .S$ $_6S \rightarrow .E$ $_6E \rightarrow .FF$ $_6F \rightarrow .7$	Scan $_6F \rightarrow 7.$ Comp. $_6E \rightarrow F.F$ Pred. $_7F \rightarrow .FF$ $_7F \rightarrow .8$	Scan $_7F \rightarrow 8.$ Comp. $_7F \rightarrow F.F$ Pred. $_8F \rightarrow .FF$ $_8F \rightarrow .9$	Scan $_8F \rightarrow 9.$ Comp. $_8F \rightarrow F.F$ Pred. $_9F \rightarrow .FF$ $_9F \rightarrow .S$ $_9S \rightarrow .G$ $_9G \rightarrow .HH$ $_9H \rightarrow .10$	Scan $_{10}H \rightarrow 10.$ Comp. $_9G \rightarrow H.H$ Pred. $_{10}H \rightarrow .HH$ $_{10}H \rightarrow .11$	Scan $_{11}H \rightarrow 11.$ Comp. $_{10}H \rightarrow H.H$ Pred. $_{11}H \rightarrow .12$	Scan $_{12}H \rightarrow 12.$ Comp. $_{10}H \rightarrow HH.$ $_9G \rightarrow HH.$ $_9S \rightarrow G.$ $_9F \rightarrow S.$ $_9F \rightarrow FF.$ $_8F \rightarrow FF.$ $_7F \rightarrow FF.$ $_6E \rightarrow FF.$ $_6S \rightarrow E.$ $_6D \rightarrow S.$ $_6D \rightarrow DD.$ $_5D \rightarrow DD.$ $_4D \rightarrow DD.$ $_3C \rightarrow DD.$ $_3S \rightarrow C.$ $_3B \rightarrow S.$ $_2B \rightarrow BB.$ $_1B \rightarrow BB.$ $_0A \rightarrow BB.$ $_0S \rightarrow A.$ $_0$ .S