The Power of Words: Generating PowerShell Attacks from Natural Language

Pietro Liguori, Christian Marescalco, Roberto Natella,
Vittorio Orbinato, Luciano Pianese

DIETI

Università degli Studi di Napoli Federico II, Italy

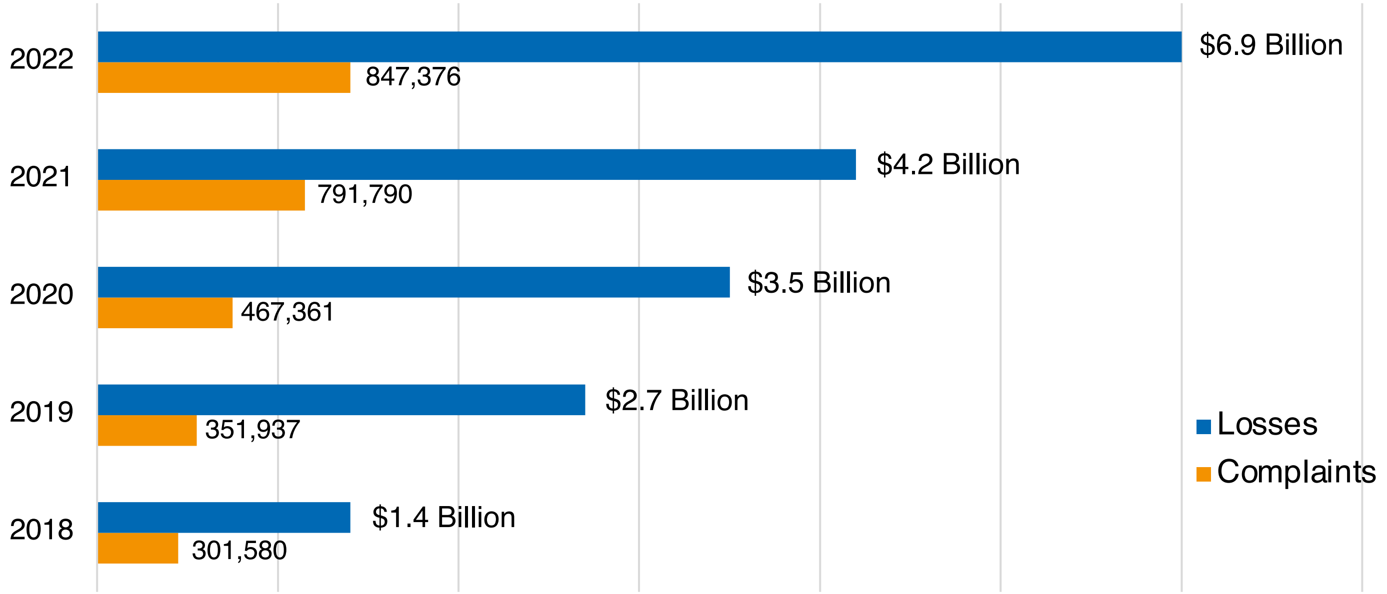
{pietro.liguori, roberto.natella, vittorio.orbinato, luciano.pianese}@unina.it

c.marescalco@studenti.unina.it

Advanced Persistent Threats

Cyber-attacks have been growing in numbers over the last decade around the world and become more sophisticated, stealthy, multi-vector, and multi-stage.
In this scenario, **Advanced Persistent Threats** (APTs) represent the most dangerous threat actor.

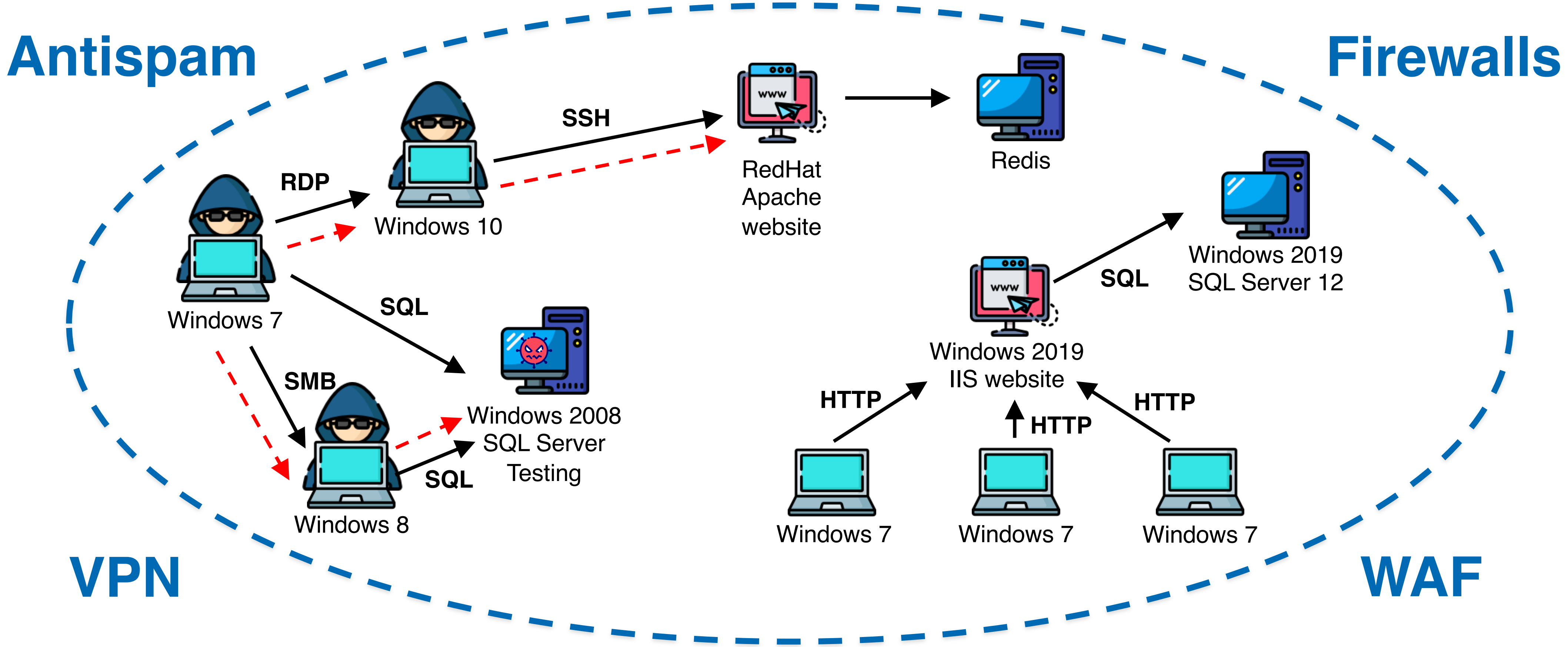
Complaints and Losses over the Last Five Years[1]



[1] Data extracted from the FBI Internet Crime Report 2022

Offensive Security

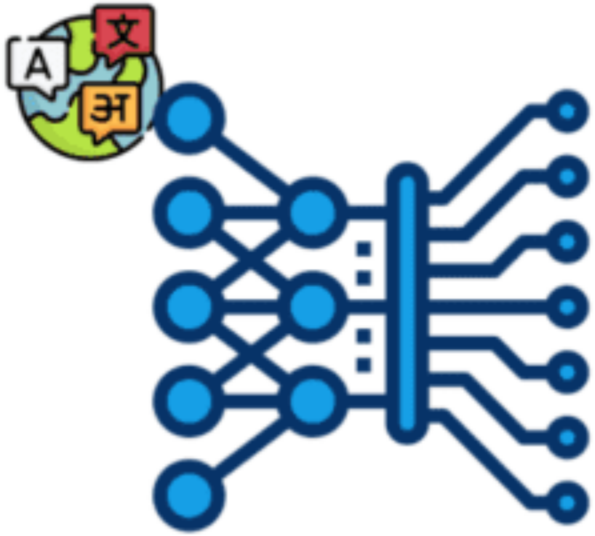
Offensive security practices, such as red teaming and adversary emulation, play a crucial role in understanding how attackers operate and how to mitigate attacks.



LLMs for Security

- LLMs can assist security analysts at writing custom software.
- NMT (Neural Machine Translation): generates code from natural language.

*“Create a **socket**, **connect to tgtHost on tgtPort**, **send the message ’ViolentPython’**, receive the response in result”*

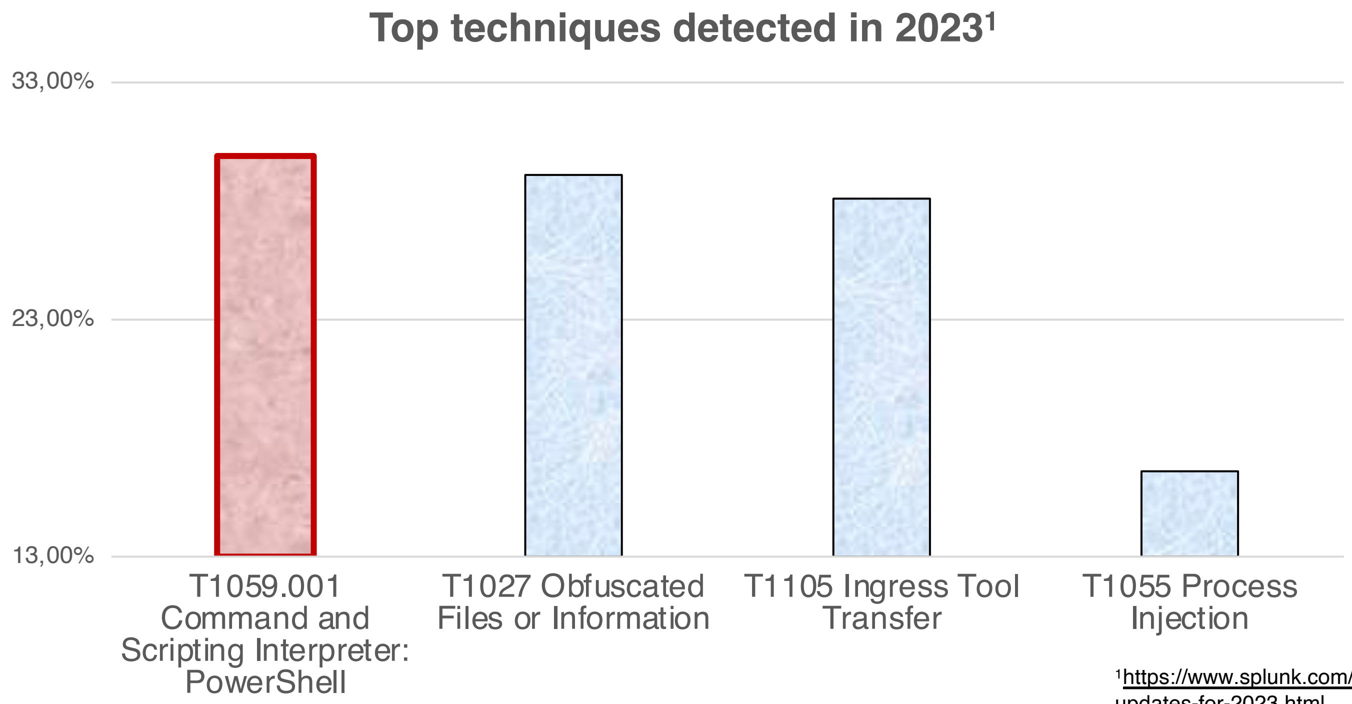



```
import optparse
import socket
from socket import *
def connScan(tgtHost, tgtPort):
    try:
        connSkt = socket(AF_INET, SOCK_STREAM)
        connSkt.connect((tgtHost, tgtPort))
        connSkt.send('ViolentPython\r\n')
        results = connSkt.recv(100)
        print '[+]%d/tcp open'% tgtPort
        print '[+] ' + str(results)
        connSkt.close()
    except:
        print '[-]%d/tcp closed'% tgtPort
```

Offensive PowerShell code generation

The **PowerShell** language is a key tool for cybersecurity in Windows systems.

We investigate NMT models to **translate natural language (NL) descriptions into PowerShell code** that implements APT behavior.



[1]https://www.splunk.com/en_us/blog/security/revisiting-the-big-picture-macro-level-att-ck-updates-for-2023.html

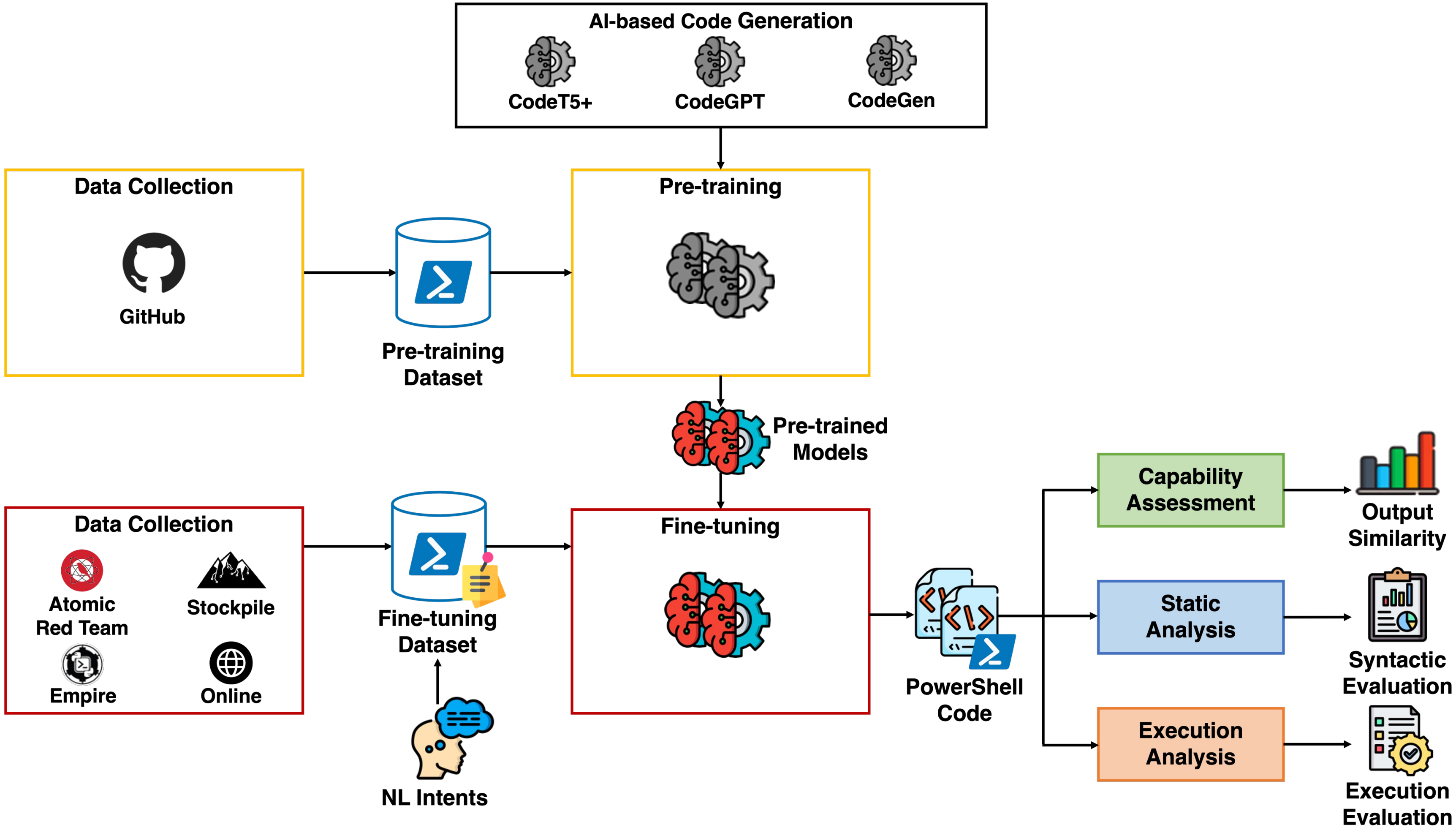
Methodology

We analyze different **strategies for data collection and training**:

- ❑ Pre-training
- ❑ Fine-tuning

Multiple **quantitative evaluations**:

- ❑ Output similarity evaluation
- ❑ Syntactic evaluation
- ❑ Execution evaluation



Pre-training

We built a dataset of **unlabeled PowerShell code**, to pre-train LLMs with:

- Masked Language Modeling (MLM)
- Causal Language Modeling (CLM)

~90k samples of **general-purpose scripts** (not limited to security) from GitHub.

```
$timestamp = Get-Date -Format s | ForEach-Object { $_ -replace ":", "." }
Import-Module ActiveDirectory

$DomainController = Get-ADDomainController| Select-Object Name
$SearchBase = Read-Host "Enter searchbase (e.g.- OU=Servers,DC=Domain,DC=local)"
$ServerList = Get-ADComputer -Filter 'operatingsystem -like "*server*" -and enabled -eq "true"' -SearchBase $SearchBase |

$Array = @()
foreach($ServerObject in $ServerList){
$colItems = Invoke-Command -ComputerName $ServerObject.Name -scriptblock {Get-ChildItem -Path Cert:\LocalMachine -Recurse

    foreach ($Server in $colItems){
        $Array += New-Object PsObject -Property ([ordered]@{
            'Server' = $ServerObject.DnsHostName
            'FriendlyName' = $Server.FriendlyName
            'Issuer' = $Server.Issuer
            'NotAfter' = $Server.NotAfter
            'NotBefore' = $Server.NotBefore
            'SerialNumber' = $Server.SerialNumber
            'Thumbprint' = $Server.Thumbprint
            'DnsNameList' = $Server.DnsNameList
            'Subject' = $Server.Subject
            'PSPath' = $Server.PSPath
            'Version' = $Server.Version})
      }
    }
$Array | export-csv C:\Temp\Certificate_Report_$timestamp.csv -NoTypeInformation
Write-Host "Export available at C:\Temp\Certificate_Report_$timestamp.csv" -ForegroundColor Cyan
```

Fine-tuning

For fine-tuning, we developed another dataset with **offensive PowerShell code**.

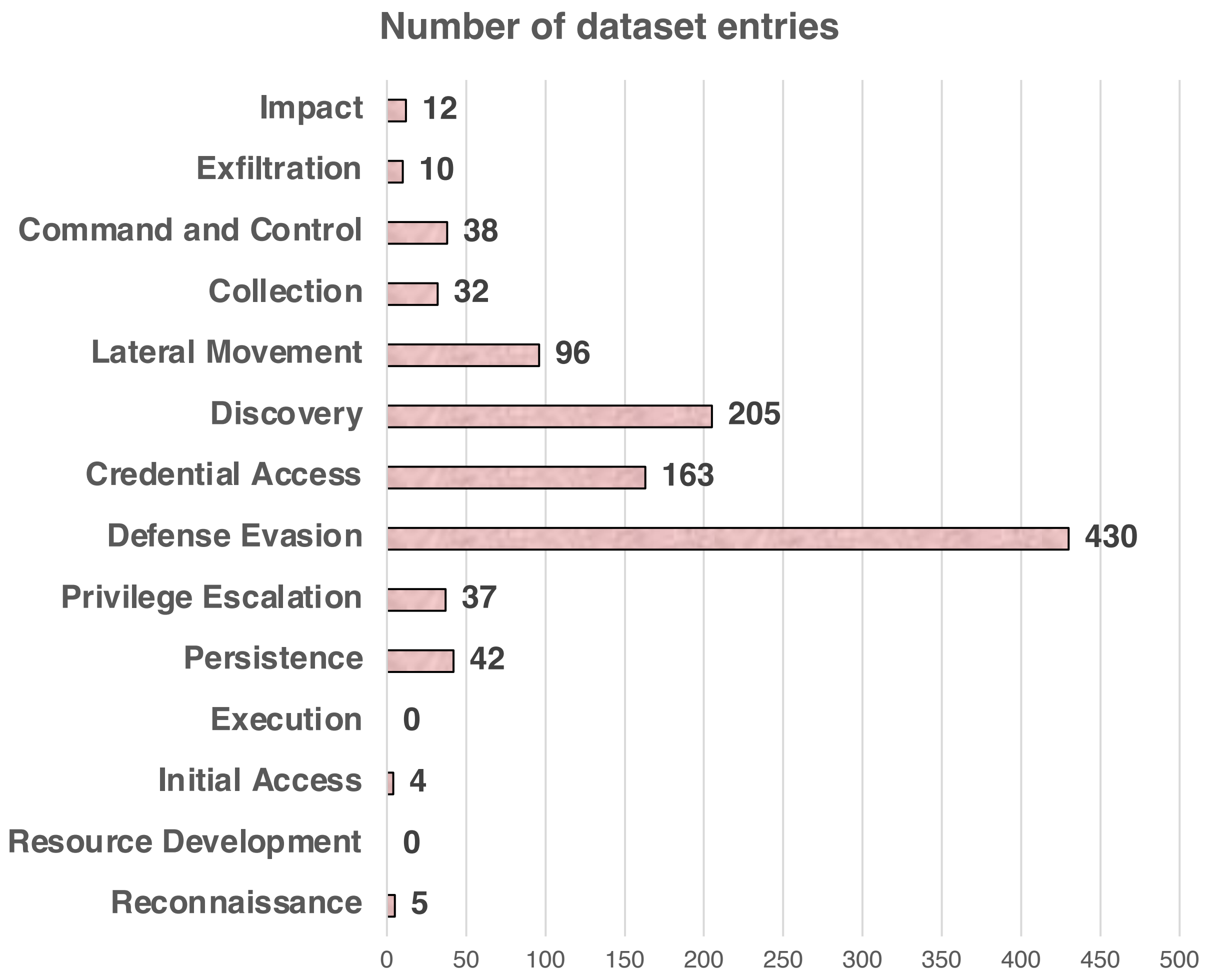
Each sample includes a pair:

- **natural language description** ("intent")
- **corresponding PowerShell code**

Data from **offensive security tools** (Atomic Red Team, Stockpile, Empire) and **public knowledge bases** (e.g., Hacktricks)

```yaml
- id: 3aad5312-d48b-4206-9de4-39866c12e60f
  name: Credentials in Registry - HKCU
  description: Search for possible credentials stored in Registry
  tactic: credential-access
  technique:
    attack_id: T1552.002
    name: "Unsecured Credentials: Credentials in Registry"
  platforms:
    windows:
      psh:
        command: |
          reg query HKCU /f password /t REG_SZ /s
```

Fine-tuning - dataset



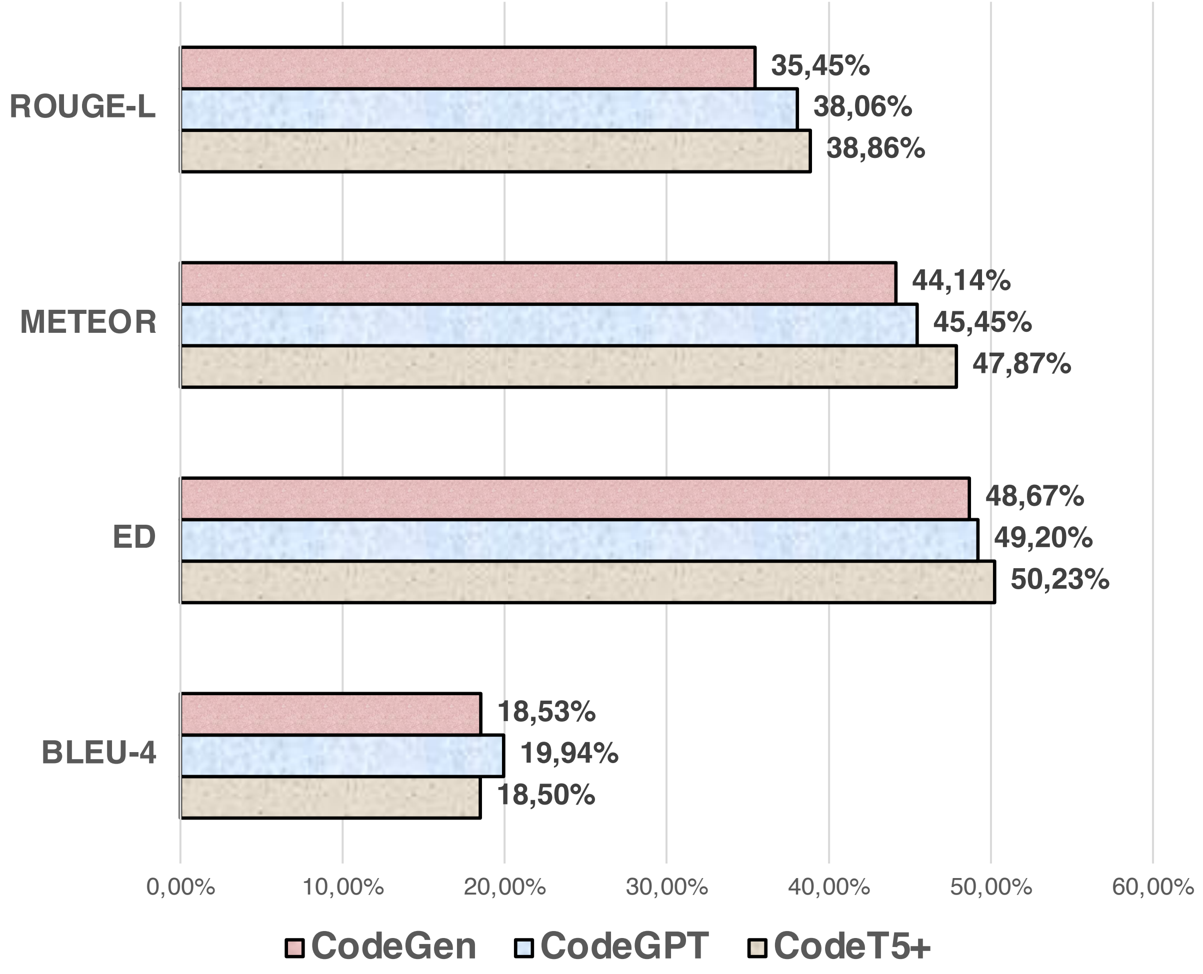
Output Similarity

Output similarity compares token-by-token the **generated code** with the code from the **ground truth**.

Specifically, we selected:

- Bilingual Evaluation Understudy (BLEU)
- Edit Distance (ED)
- METEOR
- ROUGE-L

For context (state-of-the-art code generation):
BLEU-4 = 21% and **METEOR = 38%** for Python
BLEU-4 = 25% and **ED = 44%** for UNIX shell code



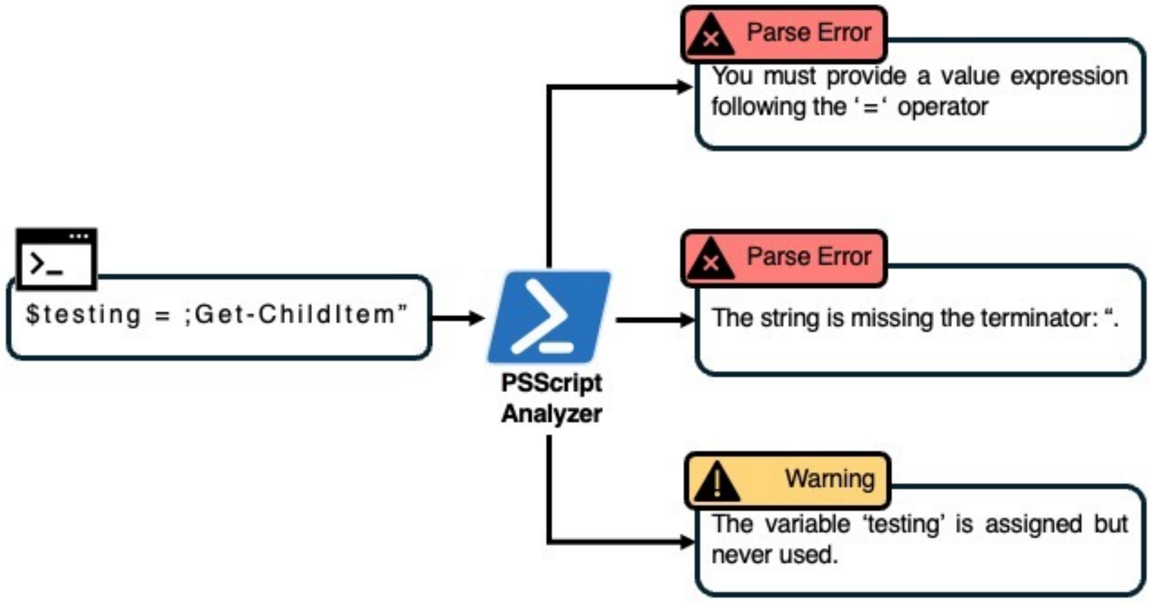
Illustrative Examples

NL Intent	Ground Truth	Model Output
Invoke compiled HTML Help (CHM) file with specific storage handler, HH file path, and CHM file path.	Invoke-ATHCompiledHelp -InfoTechStorageHandler $ { infotech_storage_handler } -HHFilePath $ { hh_file_path } -CHMFilePath $ { chm_file_path }	Invoke-ATHCompiled**HTML**Help ~~-InfoTechStorageHandler $ { infotech_storage_handler }~~ -HHFilePath $ { hh_file_path } -CHMFilePath $ { chm_file_path }
Invoke-Mimikatz cmdlet with bypassed execution policy.	powershell.exe -exec bypass -c " Invoke-Mimikatz "	powershell.exe -ExecutionPolicy Bypass -Command " Invoke-Mimikatz "

Syntactic Analysis

We evaluated the adherence to **PowerShell syntax** and compliance with **common PowerShell conventions**.

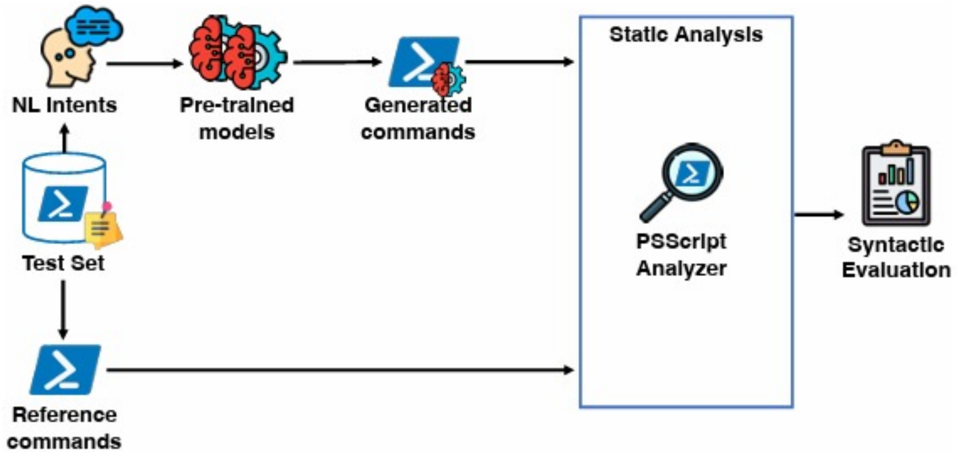
We adopt **static analyzers** to identify Warnings and Parse Errors.

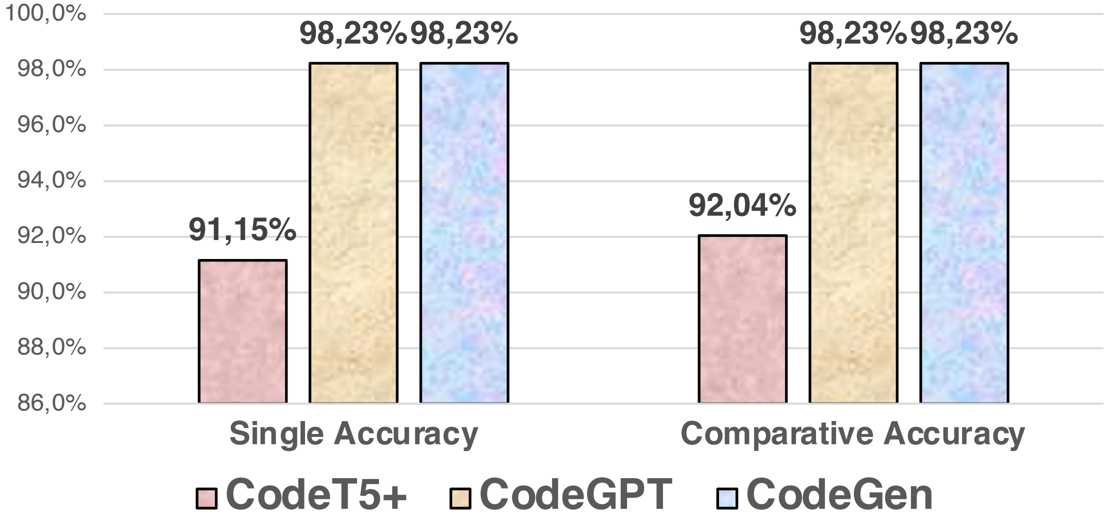


Syntactic Analysis - results

For this purpose, we leveraged two metrics:

- *Single Syntax Accuracy*: the percentage of outputs **without parse errors**.
- *Comparative Syntax Accuracy*: the percentage of outputs that **do not deviate from conventions** of the best practices.

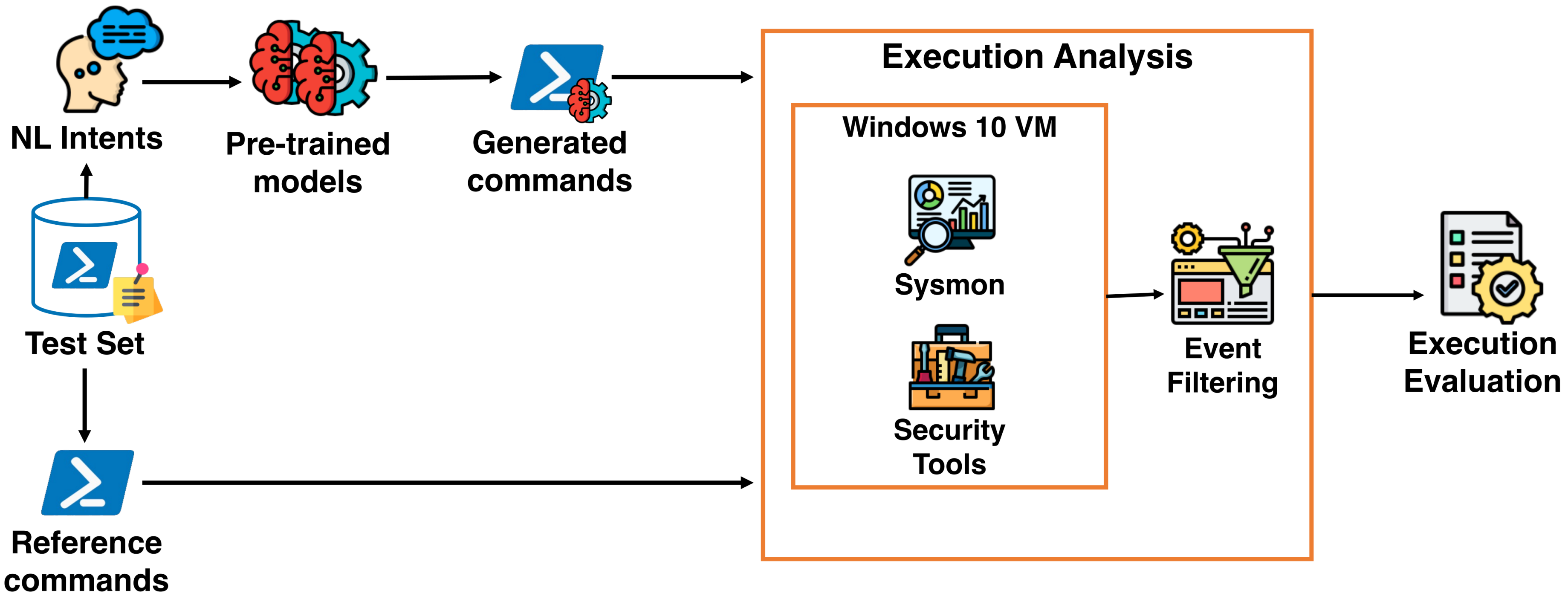




Execution Analysis

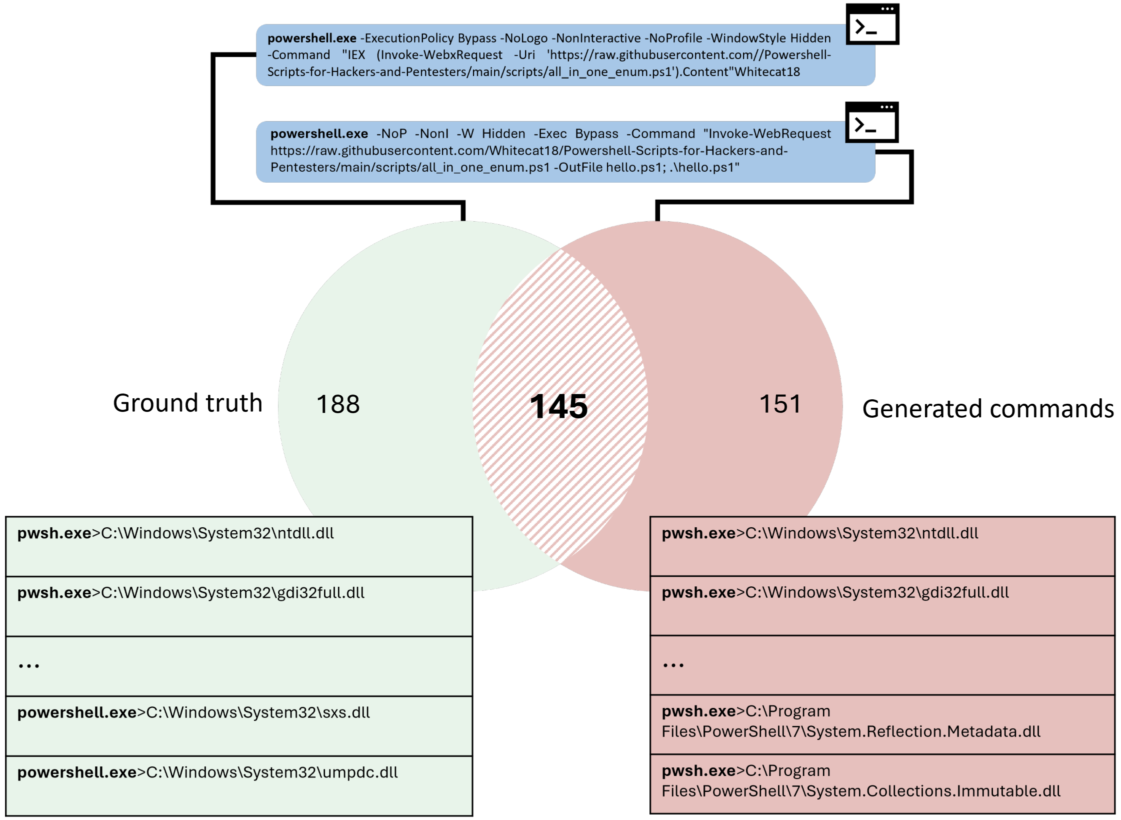
The execution analysis evaluates the generated code at **run-time**.

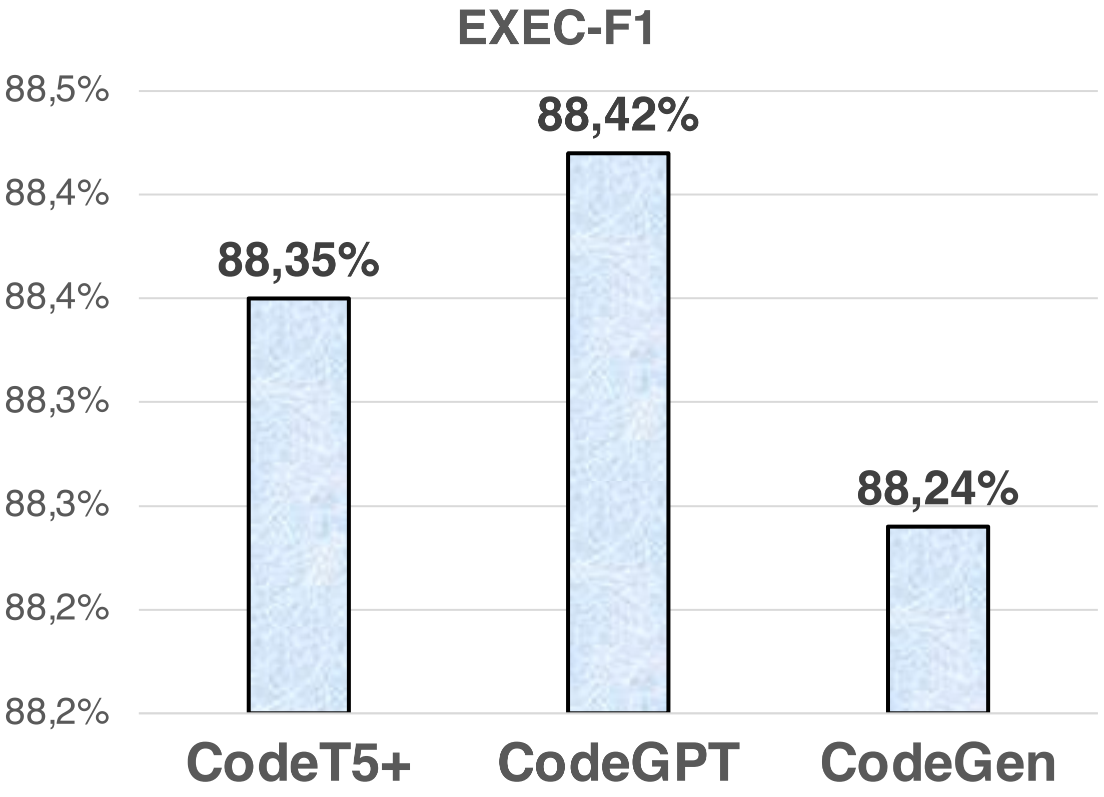
We analyze the behavior in terms of **events** occurring in the system.



Execution Analysis - results

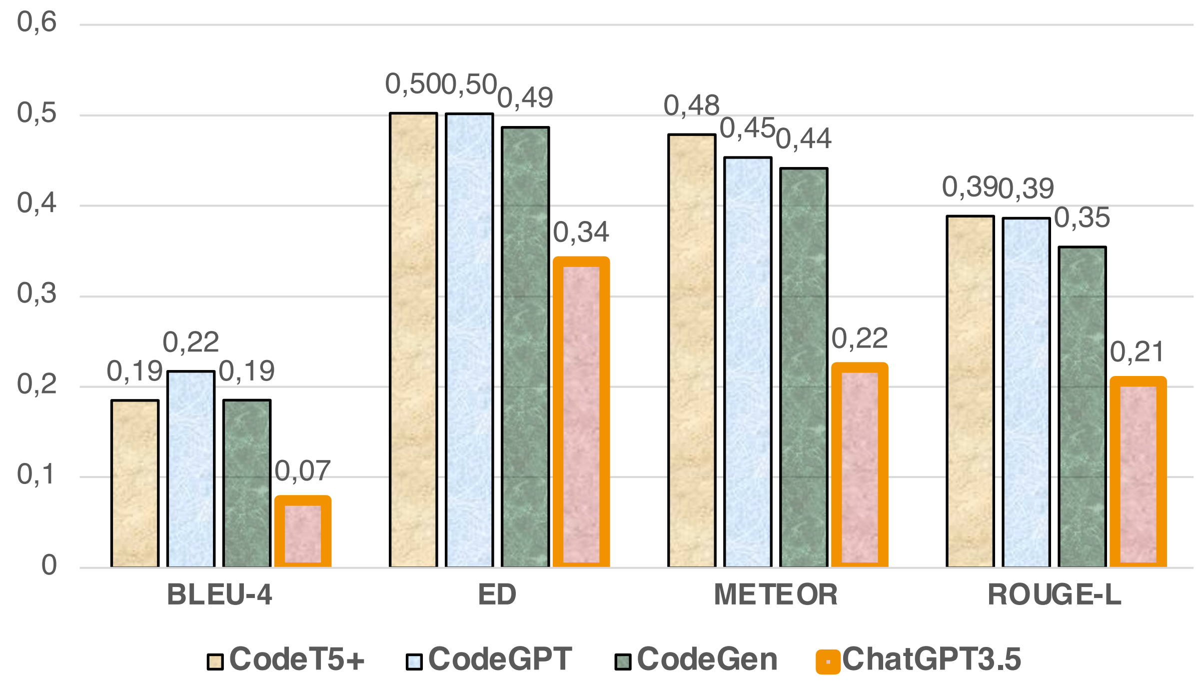
We compared the generated events using an F1-score-like measure. We conducted the analysis using a sample of 35 entries.

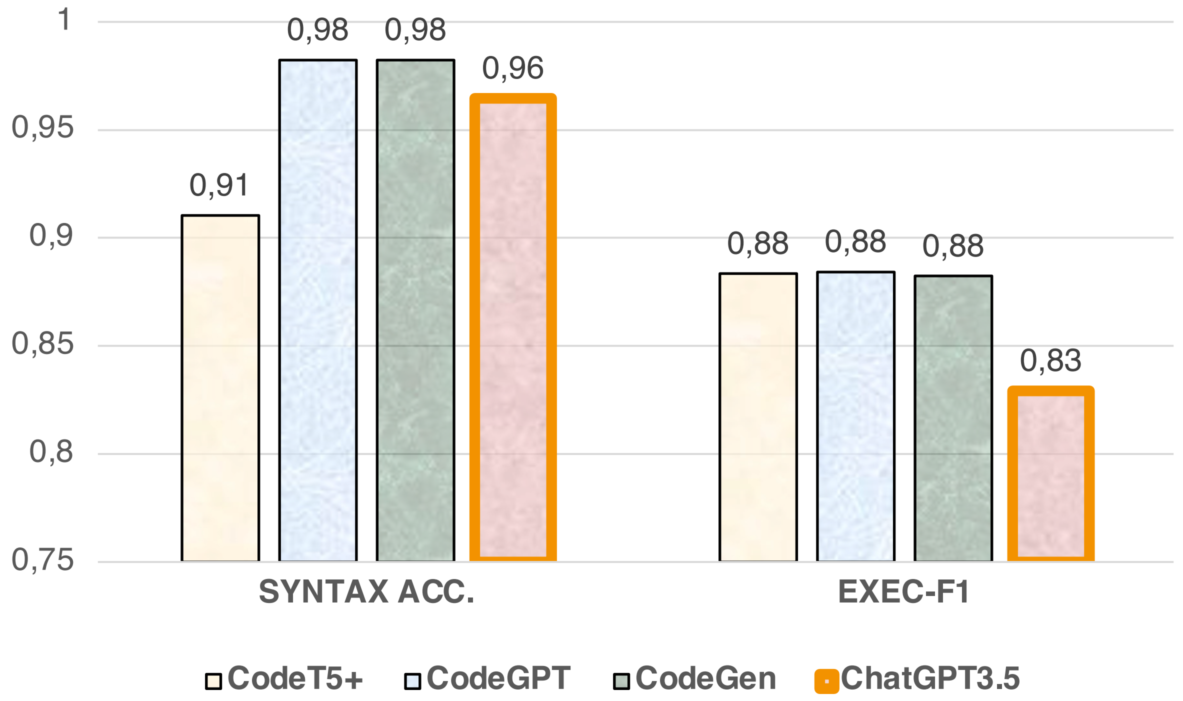




Comparison with ChatGPT

We compared the results obtained by the NMT models with **ChatGPT 3.5**, a widely-used model, not specifically designed for generating offensive code.





Conclusions

- AI-based code generators are indeed fit to generate offensive PowerShell code, achieving significant performance on this task, for all types of evaluation.

- Domain-specific fine-tuning allows NMT models to outperform state-of-the-art privately fine-tuned models, i.e., ChatGPT.

- Future work includes further analysis of the generated code, such as sandbox execution of the offensive scripts, to understand whether the code can evade detection measures, and generation of complete attack flows.

Thank you!