REALISTIC USER BEHAVIOUR SIMULATION TO COUNTER SANDBOX-EVASIVE MALWARE

Dias M.A.S.S.A

(IT21261664)

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of Science (Hons) in Information Technology Specialized in Cyber Security

Department of Computer Systems Engineering

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

DECLERATION

"I declare that this is my own work, and this dissertation does not incorporate without acknowledgement any material previously submitted for a degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books)."

11/04/2025

Signature Date

ABSTRACT

Modern malware often incorporates sandbox-evasive techniques that allow it to remain inactive or exhibit benign behaviour when executed in controlled environments lacking real user interaction. This research presents a novel defence strategy that integrates generative user behaviour simulation into sandbox environments to counter such evasive threats. The proposed system introduces a modular pipeline that begins by generating synthetic user profiles using a WGAN-GP (Wasserstein Generative Adversarial Network with Gradient Penalty), trained on behavioural metrics such as keystroke dynamics and mouse movement patterns.

To enrich these profiles, a large language model (OpenAI) is used to add interest-driven context, including search terms and URLs, resulting in realistic and varied user identities. These profiles are passed into a scripting engine that injects dynamic behavioural characteristics such as scrolling, typing, and mouse movement into predefined activity templates. A timetable engine then schedules the execution of these scripts, producing a final simulation package that is deployed into a sandbox environment.

The testing findings indicate that no user activity will allow sandbox-aware malware to suppress its payload, while the mimicking of user activity triggers full malicious execution. The system demonstrated the ability to increase malware detection efficacy through realistic human emulation, leading evasive malware to reveal its actual behaviour. The research demonstrates the viability of unifying generative models and behavioural scripting techniques to improve dynamic analysis, bolstering malware detection frameworks against contemporary evasion techniques.

ACKNOWLEDGEMENT

I would like to extend my heartfelt gratitude to **Mr. Amila Senarathne**, Senior Lecturer and Head of the Industry Engagement Unit at the Sri Lanka Institute of Information Technology, for his invaluable guidance, constant encouragement, and thoughtful supervision throughout the course of this research. His expertise and mentorship played a vital role in shaping the direction and quality of this study.

I am equally grateful to my co-supervisor, **Mr. Deemantha Siriwardana**, Assistant Lecturer, for his continuous support, constructive feedback, and timely insights that helped me overcome various challenges during this project.

A special word of thanks goes to our external supervisor, **Ms. Chethana Liyanapathirana**, Assistant Professor at the Department of Mathematics, Computer Science, and Digital Forensics, Commonwealth University of Pennsylvania, for her valuable contributions, external perspectives, and expert advice which greatly enriched the research process.

I also wish to sincerely acknowledge the **Department of Computer Systems Engineering at Sri Lanka Institute of Information Technology (SLIIT)** for providing the academic environment and necessary resources to carry out this research successfully.

Moreover, I would like to express my appreciation to my project group members for their collaboration, commitment, and shared efforts throughout the research journey. Their support and teamwork were essential to the successful completion of this project.

Finally, I thank everyone who supported this endeavor, both directly and indirectly, and contributed to the successful completion of this thesis.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
AI	Artificial Intelligence
API	Application Programming Interface
GAN	Generative Adversarial Network
WGAN-GP	Wasserstein Generative Adversarial Network with Gradient Penalty
LLM	Large Language Model
RL	Reinforcement Learning
SOC	Security Operations Centre
CSV	Comma-Separated Values
VM	Virtual Machine
JSON	JavaScript Object Notation
KS Test	Kolmogorov–Smirnov Test
MSE	Mean Squared Error
APT	Advanced Persistent Threat
UI	User Interface
OS	Operating System

1. INTRODUCTION

Malware is a significant threat to cybersecurity. It has evolved to utilize advanced technology, circumventing traditional detection systems [1]. An essential method to comprehend and mitigate these threats is malware analysis. It involves examining the malware's behavior and operational characteristics [1][2].

Malware analysis can primarily be conducted through two methodologies: static and dynamic analysis. Static analysis scrutinizes the code of the malware without executing it. In contrast, dynamic analysis monitors the activities of malware within a controlled environment [2]. These restricted environments are frequently termed sandboxes. A sandbox is an isolated virtual environment in which malware can be safely executed and monitored [1].

Dynamic analysis provides insight into the interactions between malware, the network, and the system. These are detailing that static analysis typically cannot identify [1][2]. However, the emergence of sandbox-evasive malware diminishes the reliability of dynamic analysis. These malware is engineered to identify sandbox environments and alters its behavior to avoid detection [1][3].

Common evasion tactics include identifying virtual machine attributes, delaying execution, or detecting genuine user behavior. One of the methodologies is seeking human-like interactions [3]. Malware frequently monitors activities such as mouse movements, typing, program switching, or internet browsing. If these operations appear odd or insufficient, the malware may suspect it is inside a sandbox. It can subsequently halt operations or obscure its malicious behaviours [1][3].

This study proposes an approach to counteract various evasion techniques. It introduces a technique that produces dynamic and realistic user behavior within the sandbox. The technology aims to convince the malware that it is operating on an actual user environment by replicating human-like behaviors. This improves the efficacy of

dynamic analysis and facilitates the activation of the malware's true behavior.

This chapter establishes the foundation for the investigation. The document commences with an overview of sandbox-evasive malware and the tactics it utilizes to avoid detection. It subsequently discusses how malware identifies fake environments by behavioral indicators. The limitations of static sandboxing are also addressed. The literature review examines contemporary research on user behavior emulation, innovative machine learning methodologies, and the application of generative models like GANs in cybersecurity. The chapter ultimately delineates the research gap, situates this effort within it, and articulates the objectives, scope, significance, and scientific contributions of the project.

1.1. Background & Literature Review

Fast improvements in attack and defense strategies have come from the continuous struggle between malware writers and cybersecurity experts. One of the most widely utilized techniques sandboxes are dynamic malware analysis [1]. It provides a safe environment to run suspicious files and monitor their activity without compromising real computers. Sandboxing is hence a helpful tool for modern cybersecurity [1].

But as sandboxing evolved, authors of malware also changed. Currently, many malware variants are designed to locate sandbox environments. Once found, the malware can alter its behavior or stop running to hide under inspection [1][2].

This type of malware is called as sandbox-evasive malware. It searches for relics from virtualization, missing user interactions, or other environmental cues [2]. If the malware find proof of it not running on a valid user PC, it may remain inactive or behave differently to hide itself [1][2].

This makes the boundaries of current sandbox techniques absolutely obvious. Right now, it is needed to raise the realism of sandbox environments in order to combat against sandbox-evasive malware. One significant advance is simulation of natural

user behavior [2][3]. Whether it is in a real system, malware largely depends on the presence of actions such mouse movement, typing, and program usage. Therefore adding user behaviour to the sandbox can make it hard for the malware identify whether it is in a sandbox or not [2][3].

This overview of the literature investigates the techniques used to find and avoid sandboxes. It also considers the challenges creating sensible user behavior. It also addresses the use of machine learning especially Generative Adversarial Networks (GANs) in Cyber Security field. The objective is to understand current studies and identify new opportunities to enhance sandbox-based analysis.

1.1.1 Introduction to Sandbox Evasive Malware

In cybersecurity, particularly in dynamic malware analysis, sandbox environments are essential. These configurations provide a controlled, isolated environment in which potentially malicious activities can be executed safely [1]. This enables specialists to observe malware behavior without risking real systems or data. Through sandboxing, researchers can observe malware interactions with the operating system, files, registry, and network services [1].

This behavioral awareness is essential for understanding, coordinating, and mitigating risks. It also enables the identification of zero-day or unknown malware that may bypass traditional signature-based techniques [1]. However, as sandbox systems have progressed, so have the defenses developed by malware authors [1][2].

Contemporary malware frequently incorporates mechanisms to detect virtualized or sandboxed environments. Upon detection, the malware may alter its behavior to obscure its malicious actions. This method, known as sandbox evasion, has become one of the most formidable challenges in dynamic malware research [1][2].

Time-based evasion is a frequently employed evasion strategy. This strategy involves malware deliberately reducing its propagation speed to evade detection under the

sandbox's monitoring parameters [1]. Sandboxes typically function for a predetermined duration; if the malware remains inactive throughout this interval, it may become undetectable. Another strategy is environment scanning, in which malware detects indications to figure out whether it is functioning on a virtual or physical system [1][2][3].

These checks may encompass the examination of certain files, registry keys, active programs, or system drivers, specifically about their presence or absence. Numerous signs are exclusively located on genuine machines, rather than in simulated environments [2]. If the anticipated indicators are lacking, the malware may either reduce its payload or simulate benign behavior to deceive the inquiry [1][2].

The range and complexity of sandbox evasion techniques continue to expand. A thorough examination of these strategies is presented in the investigations conducted by Mohanta and Saldanha in Malware Analysis and Detection Engineering. Their findings emphasize the pressing necessity for continuous advancement in sandbox systems to accommodate the rapid evolution of threats [1].

The authors of Defining Sandbox Evasion: How to Increase the Successful Emulation Rate in Virtual Environments emphasize the necessity of constructing highly realistic simulation environments. These may diminish the effectiveness of time-sensitive and context-aware detection techniques utilized by malware [3]. The research proposes advanced virtualization approaches and interactive modeling to enhance emulation success rates [3].

The paper Detecting Environment-Sensitive Malware addresses the challenge of identifying malware that exhibits variable behavior based on its execution environment [4]. It illustrates how certain malware samples alter their behavior or remain dormant when encountering a sandbox environment. This renders classical dynamic analysis increasingly difficult [4].

Static detection alone is insufficient, as malware is very responsive to its environment.

Sandboxes must now emulate not only technical system conditions but also authentic human interactions. These may include mouse movements, keyboard activity, and file or application usage indicators that would convince the malware operating on an actual user PC [1][2].

1.1.2 Anti-Evasion Strategies

To counter increasingly evasive malware, researchers have developed several strategies aimed at improving the reliability of sandbox environments. These strategies focus on encouraging malware to execute its true functionality by reducing its ability to detect artificial or controlled settings [1]. By doing so, analysts can observe malicious actions more effectively and prevent potential harm [1][2].

One widely researched method involves dynamically modifying the sandbox environment. This approach aims to disrupt malware's ability to detect whether it is running in a sandbox or a real system [5]. It works by frequently changing certain system characteristics during analysis, making it harder for malware to identify consistent sandbox traits [5].

For example, the study Investigating Anti-Evasion Malware Triggers Using Automated Sandbox Reconfiguration Techniques explores how changing file structures, network settings, and user activity patterns can help [5]. By presenting a constantly evolving system profile, the sandbox becomes more convincing. This reduces the chance of detection and increases the likelihood of observing real malware behaviour [5].

Another effective technique is environment imitation. This strategy attempts to mimic the appearance and activity of a real user environment. It includes launching typical applications, generating logs, and browsing websites actions that reflect how a real person uses a system [6]. The paper Impersonating a Sandbox Against Evasive Malware discusses how this method deceives malware into executing its payload by simulating ordinary behaviour patterns [6].

When malware sees familiar system behaviour, it is less likely to suspect that it is being analysed. This allows it to expose its actual functionality, enabling accurate analysis [6]. Such imitation helps sandboxes capture the true intent of malware in a way that static configurations cannot [6].

In addition, a study titled MalwareEnvFaker introduces a technique for improving Linux-based sandbox environments [7]. This method uses a combination of tracing, filtering, and emulation to replicate a complex and believable user environment. The approach creates dynamic system states that closely resemble those on real user machines [7]. The study shows that this strategy is highly effective against environment-sensitive malware. It also increases the overall robustness of sandbox analysis [7].

A more recent and novel strategy is the simulation of user behaviour within the sandbox. This technique focuses on recreating the actions and interactions of an actual user to fool malware into believing it is operating in a real environment [1]. It includes mimicking human-like input such as mouse movements, typing, or switching between applications. By doing this, the sandbox appears more authentic, prompting the malware to reveal its malicious behaviour [1][2].

These strategies collectively represent significant progress in the fight against evasive malware. They provide new ways to adapt sandbox environments, making them harder to detect and more reliable for dynamic analysis. A later section will explore the role of user behaviour simulation in greater detail, as it has become one of the most promising anti-evasion techniques.

1.1.3 Behavioural Indicators Used in Malware Detection

Modern malware often relies on behavioural indicators to determine whether it is running in a real user environment or within a sandbox. These indicators help the malware assess the presence or absence of natural human interaction. If realistic

behaviour is not observed, the malware may delay, suspend, or completely alter its execution to avoid detection [1][2].

One of the most common behavioural checks is mouse movement. Malware may monitor for random, human-like cursor activity across the screen [2]. If no movement is detected or if it follows a scripted, predictable path, the malware may assume it is inside a sandbox [2]. Similarly, keyboard input is another key signal. Malware often waits for actual keystrokes before activating its payload. Automated typing sequences or total inactivity may raise suspicion for environment-aware malware [2][3].

Another critical indicator is application usage. Malware may check whether typical user applications like browsers, word processors, or media players are being launched and interacted with [3]. If the system appears idle or only runs background services, this may reveal an artificial environment [3][4].

File system activity also plays a role. Malware may look for changes in file timestamps, recent file edits, or access to certain directories. These interactions suggest a system in daily use rather than a static or freshly reset sandbox. The absence of such activity can prompt the malware to avoid execution or enter a dormant state [4].

Some malware also evaluates window focus changes or tab switching patterns. These subtle actions are harder to fake through automation. When absent, they may indicate that the system is not being actively used by a human [4][5].

Most studies show how many malware samples rely heavily on these small behavioural clues to make execution decisions [2]. The more realistic the environment appears, the more likely the malware is to act as it would on a victim's machine. Similarly, lack of interaction significantly reduced the activation rate of sandbox-evasive malware during testing [2].

These behavioural indicators have become a core component of sandbox detection techniques. To overcome this challenge, security researchers must replicate not just

system configurations, but also real user behaviour. This includes simulating input patterns, usage timelines, and interaction habits that malware expects to see [3][5].

By addressing these behavioural gaps, sandbox environments can more effectively trick malware into exposing its true intent. This sets the stage for more advanced approaches, such as user behaviour emulation and interaction modelling, which are covered in the following sections.

1.1.4 User Behaviour Emulation in Sandboxes

Emulating user behaviour has become a core component of modern sandbox environments. This technique is designed to counter malware that uses sandbox detection to avoid exposure[1][2]. By replicating real user actions inside a virtual machine, the goal is to fool the malware into revealing its true functionality. As malware becomes more sensitive to its surroundings, traditional static sandboxes without interactive features are no longer sufficient [2][8].

Malware often checks for signs of normal user activity. It may look for typing, cursor movements, file access, or browsing behaviour. When these actions are absent, the malware can assume it is running in a sandbox. It may then remain inactive or behave in a harmless way, avoiding detection [1][8].

To tackle this, researchers have introduced user behaviour emulation into sandbox environments. The idea is to replicate realistic interaction patterns that closely resemble what a real user would do [8]. Actions such as moving the mouse, opening websites, typing into documents, or switching between apps are simulated to create a believable environment [8][9].

The importance of this approach is supported by studies like Combating Sandbox Evasion via User Behaviour Emulators and Enhancing Malware Analysis Sandboxes with Emulated User Behaviour [8][9]. These papers highlight how simulated user activity improves the chances of malware revealing its behaviour. By replicating a

natural environment, malware is more likely to execute its payload without realising it's being observed [8][9].

Several methods exist to simulate user activity. One basic technique uses pre-scripted events. For example, a script may move the mouse, open a browser, or type random text into a file at fixed intervals [1]. These tactics work for less advanced malware, but they are often too predictable. More sophisticated malware can detect repetitive or timed interactions and identify them as artificial [9].

To overcome this limitation, researchers have developed more advanced models. These systems rely on probabilistic behaviour patterns and randomised event timing to simulate human unpredictability [8][9]. Instead of fixed scripts, the sandbox dynamically generates actions based on user profiles. These profiles determine what actions should be performed, in what order, and how often.

One such model is UBER, which integrates user behaviour simulation directly into the sandbox architecture [9]. The system starts by collecting real user interaction data from normal computing environments. This data is then processed by a User Profile Generator, which builds realistic usage profiles [9]. Each profile includes patterns such as application usage, interaction timing, and common tasks.

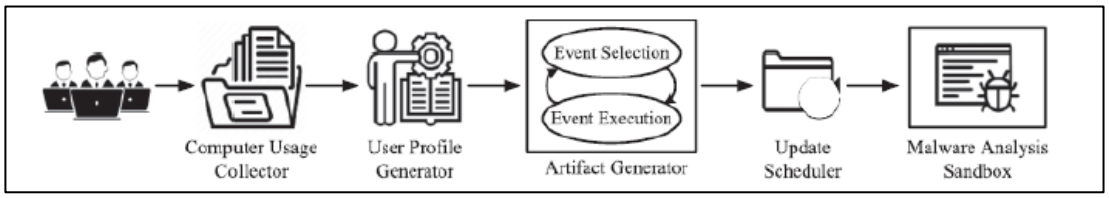


Figure 1: UBER model architecture

Source: [9]

Next, an Artifact Generator selects specific actions from the profile. It injects these into the sandbox in a probabilistic manner to ensure variability and realism [9]. This helps the system avoid repeating fixed patterns that might be flagged by malware. As

shown in the UBER architecture (Figure 1), the goal is to make user interaction appear genuine and unpredictable.

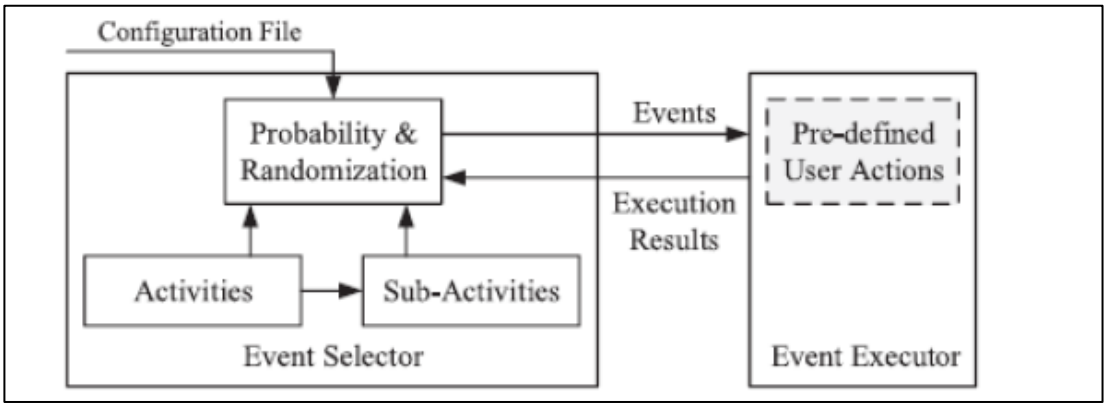


Figure 2: User behaviour generation in UBER model

Source: [9]

Further enhancing this system are the Event Selector and Event Executor components (Figure 2). The Event Selector chooses actions and sub-actions based on probabilities and rules defined in a configuration file [9]. It considers event order, timing, and context. The Event Executor then performs those actions inside the sandbox environment. This layered approach replicates user behaviour more accurately and introduces randomness that malware cannot easily anticipate [9].

Despite the effectiveness of these techniques, challenges remain. Some advanced malware can detect even small inconsistencies in behaviour timing, frequency, or sequence [9]. Repetitive or mechanical actions may still trigger detection if not handled carefully. As malware continues to evolve, so must user emulation strategies. Ongoing research is essential to enhance realism, randomness, and contextual responsiveness in sandbox simulations [8][9].

1.1.5 Advanced ML Techniques for User Behaviour Emulation

The incorporation of artificial intelligence (AI) and machine learning (ML) into cybersecurity has created new opportunities for improving the authenticity and

efficiency of sandbox environments. This is particularly true in simulating realistic user behaviour [10]. Among AI-driven techniques, large language models (LLMs) are notable for their ability to generate complex, human-like actions. This capability significantly enhances the effectiveness of sandboxes in deceiving advanced malware [10].

Language models, as discussed in User Behaviour Simulation with Large Language Models, are trained on large datasets that cover a wide range of human behaviours and linguistic structures [10]. These models can accurately predict and generate sequences that closely resemble natural language and decision-making patterns [10]. Researchers can use LLMs to reproduce various types of user behaviour in sandbox environments. These behaviours include casual web browsing, document editing, and email communication [10].

However, it is important to note that the LLMs evaluated in this study were not specifically designed to simulate computer-based user behaviours. Actions such as mouse movement, keystrokes, or file manipulation are crucial for effective human behaviour emulation in sandboxes [10]. Instead, LLMs excel in generating text-based interactions and scenarios that closely mimic real human communication and choices [10].

Although this is a limitation, LLMs still offer a substantial improvement over traditional scripted method. They provide dynamic, context-aware behaviour that can be adjusted over time to reflect new patterns in user activity [10]. This makes them a valuable addition to modern sandbox environments, even if their role is focused on higher-level behavioural simulation rather than low-level interaction emulation [10].

1.1.6 Current Usage of GANs in Malware Detection & Prevention

In the field of machine learning, generative adversarial networks (GANs) signify a significant advancement. Their ability to generate synthetic data that closely resembles real-world data is widely recognized [11]. Ian Goodfellow introduced Generative

Adversarial Networks (GANs) in 2014, which consist of two neural networks: the generator and the discriminator [11][12]. Adversarial learning is the technique employed in the collective training of these networks. The discriminator distinguishes between authentic and synthetic data, whereas the generator produces fake data. This process continues until the generator produces data that the discriminator cannot identify as fake. Consequently, GANs serve as a formidable tool for applications such as image generation, text generation, and cybersecurity [11][12].

Cybersecurity researchers have mostly focused on improving malware identification and classification through the utilization of GANs. _Generative Adversarial Network for Enhancing Deep Learning Based Malware Classification_ examines the potential of GANs to improve classification models [11]. GANs facilitate the augmentation of training datasets available for deep learning systems by generating synthetic malware samples. This enhances the generalization capability and accuracy of malware detection models [11].

GANs possess a primary advantage in their ability to generate diverse and representative samples. In cybersecurity, where various forms of malware diminish the efficacy of traditional detection methods, this is particularly advantageous. The development of synthetic samples enables models to learn from a broader spectrum of assault patterns, hence augmenting their robustness [12].

A comprehensive examination of GAN application in malware research is presented in the publication Generative Adversarial Networks for Malware Detection: A Survey[12]. It delineates their application across several fields, including adversarial example generation, malware classification, and evasion attack identification. The study highlights the adaptability of GANs in tackling complex tasks related to malware identification and prevention [12].

These findings suggest that GANs can simulate adversarial settings, hence allowing security systems to train under progressively challenging conditions. This enhances their efficacy in confronting real threats [12].

Despite these advancements, minimal research has been conducted on the generation of user profiles in sandbox environments with GANs. Instead, than emulating rational user behavior to counter sandbox-evasive malware, the majority of contemporary efforts focus on improving malware detection and classification [11]. GANs are an optimal selection for this task because to their principal advantage: the ability to generate realistic, high-quality data.

Synthetic user profiles generated by GANs may facilitate the creation of dynamic and diverse user behavior. This would facilitate the activation of evasive malware samples and enhance the authenticity of sandbox settings.

1.2 Positioning the Research

This study seeks to address a significant gap in the field of malware analysis. The primary objective is to develop authentic user behavior profiles to enhance the usefulness of the sandbox environment. Prior research has examined several techniques to evade sandbox detection and replicate human interaction [1][8][9]. Despite the inclusion of perceptive analysis, these projects have been largely ineffective in generating dynamic and diverse user profiles that can consistently deceive advanced malware.

This research try to address this limitation. The study employs advanced machine learning techniques, particularly Generative Adversarial Networks (GANs), to accurately simulate plausible user behavior. Enhanced simulations can optimize sandbox environments and increase the likelihood of detecting evasive malware [8][9].

This targeted approach positions the initiative as a direct response to a major issue in modern malware analysis.

1.3 Research Gap

Previous research has made notable progress in simulating user actions within sandbox environments. These efforts aim to deceive malware that relies on sandbox detection techniques. One of the most well-known contributions is the UBER model. It simulates user activities to create a more realistic environment for malware analysis [8][9]. However, most of these approaches rely on pre-scripted or randomly generated user actions. Over time, these actions may become predictable. This predictability limits their effectiveness against advanced malware capable of adapting and recognising artificial patterns [8][9].

There is a clear need to develop more realistic and dynamic user behaviour simulators. These simulators should not only imitate general activity but also introduce variability that resembles real-world human behaviour. Without this, sophisticated malware can still identify artificial environments and avoid detection [8][9].

Another limitation in current techniques is the use of static data sources. For instance, some systems rely on public trend data, such as Google Trends, to simulate user behaviours like browsing history or bookmarks [8][9]. While this provides a useful starting point, it does not account for the context-specific and evolving nature of real user interaction. As a result, these simulations fail to reflect the complexity of human activity and may expose the sandbox to detection [8][9].

Recent developments in Large Language Models (LLMs) have shown strong potential for simulating human-like behaviour. However, their applications have mainly focused on generating conversation and decision-making scenarios [10]. These models are not designed for reproducing system-level actions such as typing, moving the mouse, or interacting with software tools. While their narrative and contextual output is useful, they cannot fully replicate the physical user interactions required in sandbox environments [10].

Generative models, particularly those based on adversarial learning, have already shown success in fields like image generation and text synthesis. These models are capable of producing high-quality, complex data patterns [11]. Despite their proven value in other domains, their application in simulating user behaviour within sandbox environments remains largely unexplored [11].

Most current work using generative models has concentrated on enhancing deep learning-based malware categorization. Although this is a great contribution, generative models' capacity to provide reasonable user profiles has not yet been completely utilized [11][12]. By closely copying actual human behavior, these profiles could help to increase sandbox authenticity. This would improve dynamic analysis's general accuracy and enable more efficient exposure of evasive malware [11][12].

Table 1: Research gap

Research Paper	**User Behaviour Simulation**	**Dynamic Data Utilization**	**Generative Model Usage**	**Realism of Simulated Behaviour**
Combating Sandbox Evasion via User Behaviour Emulators	✔	✕	✕	✕
Enhancing malware analysis sandboxes with emulated user behaviour	✔	✕	✕	✔
Investigating Anti-Evasion Malware Triggers Using Automated Sandbox Reconfiguration Techniques	✔	✕	✕	✕
MalwareEnvFaker A Method to Reinforce Linux Sandbox Based on Tracer Filter and Emulator against Environmental-Sensitive Malware	✔	✕	✕	✕
User Behaviour Simulation with Large Language Model	✔	✕	✔	✔
Proposed Approach	✔	✔	✔	✔

1.4 Research Problem

This study seeks to address a significant limitation identified in the literature: the lack of realistic and dynamic user behavior profiles within sandbox environments. Developing user profiles that can precisely replicate authentic user interactions is important. These profiles must possess dynamic attributes that enable adaptability to diverse situations and exhibit rational conduct during malware investigations.

The primary challenge lies in devising user profiles that accurately reflect the complexity of genuine human behavior. Often, contemporary sandbox systems rely on basic, static, or repetitive behavior simulations. Advanced malwares readily detect these fake patterns. Upon detection, the malware can alter its behavior or avoid execution to escape scrutiny.

This problem reveals a fundamental defect in current sandbox systems. The ability of malware to recognize deviations from typical behavior enables it to evade detection. A more sophisticated solution—capable of producing adaptable, rational, and dynamic user interactions—is consequently essential.

The significance of this study arises from its potential to significantly enhance the effectiveness of sandbox-based malware analysis. Bridging the gap between real user environments and simulated ones will significantly hinder malware's ability to detect the sandbox. Accurate simulation of user action increases the likelihood that evasive malware will reveal its true behavior during dynamic analysis.

This study provides a substitute for more advanced sandbox systems. Authentic user profiles developed and simulated enhance the accuracy of sandbox settings. This seeks to reduce the efficacy of sandbox evasion and enhance the overall accuracy of malware detection and prevention methods.

1.5 Research Objective

1.5.1 Main Objective

This project seeks to develop a system capable of generating realistic and dynamic user behavior profiles, specifically designed to replicate human interaction within sandbox environments. This is essential since traditional sandbox systems often fail to accurately depict the complex nature of real user interactions. Advanced malware may therefore recognize these simulated conditions and modify its behavior to avoid detection.

The technique aims to enhance the authenticity of the sandbox environment by simulating actual human behaviors, including typing, mouse movement, internet browsing, and application usage. The simulated environment must closely resemble an actual user's PC. This exposes the malicious behavior of evasive malware during examination, hence promoting its typical execution.

In contrast to conventional methods reliant on scripted or repetitive tasks, the proposed system focuses on producing adaptable and context-sensitive behavioral patterns. These activities must adapt according to the circumstances and reflect the inherent unpredictability of genuine user interactions. Maintaining this level of variation is essential, as numerous evasive malware variants detect sandboxes through routine or automated behavior.

The objective also encompasses the creation of user profiles that incorporate timing, sequencing, and decision-making elements that emulate genuine human behaviors, with the replication of surface activities. This increased level of behavioral emulation suggests that malware could use the sandbox as a genuine environment, hence improving the efficacy of dynamic analysis tools.

The research enhances detection and prevention strategies by reducing malware's ability to exploit vulnerabilities in sandbox simulations.

1.5.2 Specific Objectives

To establish a schema for user profiles: - This objective seeks to provide a methodical framework defining the main traits of user behavior. All developed profiles will be modeled by the schema, which defines variables including keystroke dynamics, mouse movement trajectories, user interests, visited URLs, and search terms connected to browser behavior. It will enable exact modeling of real user behavior and give consistency among created profiles.

To gather and preprocess user behavior data: - This objective is to compile user interaction statistics mostly related to **keystrokes and mouse movements** from publicly available sources. Preprocessing will then go over the raw data to remove superfluous features, noise, and discrepancies.

To develop and train a generative model for the synthesis of authentic user behaviors: - This aim consists in developing and training a Generative Adversarial Network (GAN) competent in producing excellent user profiles. Mouse trajectories, keypress timings, digraphs, and trigraphs will all be combined in these profiles. The generative model has to reproduce flexible and context-sensitive behavior to help the sandbox to seem like a real user-operated environment.

To create automation scripts derived from user profiles for behavioral simulation: - This objective aims to generate automation scripts from the generative model output. These scripts will copy real user behavior within the sandbox setting. Behavioral traits from the profiles will be gathered by the scripting engine and included into current activity templates. These templates will then be changed and used in line with a dynamically generated timetable, therefore guaranteeing reasonable and varied behavior over time.

To include the created scripts into a sandbox environment and emulate user behavior: - Implementing the automation scripts inside the sandbox for thorough simulation is this last objective. It includes supervising the final execution script,

scheduler, and behavioral template integration. The aim is to imitate human interaction in real-time so generating sandbox-evasive malware to carry out its hidden activities.

1.6 Scope of the Study

This research aims to improve sandbox-based malware analysis by the simulation of authentic user behavior. The project entails creating a system that generates dynamic user profiles and automation scripts to simulate authentic user interactions within a sandbox environment.

The generative model created in this study is restricted to generating only low-level behavioral data. This encompasses mouse movement, keystroke dynamics, digraphs, and trigraphs. These behaviors are essential for replicating complex user interaction that sandbox-evasive malware frequently employs as indicators of detection.

The scripting engine utilizes these created properties to produce automation scripts. These scripts correspond to revised user activity templates and are executed within the sandbox based on a dynamically generated schedule. This leads to the emulation of diverse user behaviors, encompassing:

- Mouse movement related behaviours
- Keystroke related behaviours
- Web browser related behaviours
- File operation related behaviours
- Application related behaviours

It is essential to recognize that the GAN model generates mouse movement and keystroke-related data, whereas other behaviors, including browser activity, file operations, and application usage, are managed via static activity templates. These are chosen and implemented according to the user profile and established timeline.

The research excludes conversational AI, malware classification, network analysis, and reverse engineering. The scope is confined to the realistic simulation of user behavior within a sandbox to enhance virus detection.

2. METHODOLOGY

2.1 Scope of the study

The methodology of this research consists of several key phases, each contributing to the goal of simulating realistic user behaviour in sandbox environments to detect evasive malware. The process is structured into the following main components:

- **Data Preprocessing**: - Datasets accessible to the public, comprising mouse movement and keystroke data, are utilized. The data is subjected to cleansing, normalization, and formatting to guarantee consistency and usefulness.

- **Defining the Schema**: - A structured schema is developed to represent user behaviour profiles. The schema includes key features such as digraphs, trigraphs, and mouse movement characteristics, acting as a blueprint for all generated profiles.

- **Profile Generation using GAN**: - A Generative Adversarial Network (GAN) is trained using the preprocessed data. It generates synthetic user profiles that emulate authentic low-level behaviors exhibited by actual users.

- **Profile Enrichment**: - Four arbitrary user interests are chosen for each profile. These are sent to OpenAI to produce authentic search phrases and URLs, which are incorporated into the profile to enhance realism.

- **Script Generation**: - A scripting engine processes each profile and modifies established behavior templates. It subsequently produces automation scripts to replicate user behavior within the sandbox environment.

- **Timetable Scheduling**: - A timetable engine generates a plausible activity schedule. It designates particular behaviors and allocates them to distinct time intervals to replicate continuous and diverse user actions.

- **Sandbox Integration**: - The completed automation script, encompassing the scheduled actions and revised templates, is executed in Cuckoo Sandbox. This replicates real-time user behavior and incites sandbox-evasive malware to activate.

2.2 System Architecture

The system architecture of the proposed solution is designed to simulate realistic user behaviour within a sandbox environment in order to counter sandbox-evasive malware. It is composed of modular and interconnected components that enable the end-to-end generation, management, and simulation of synthetic user activity. Figure 3 illustrates the overall structure of the system.

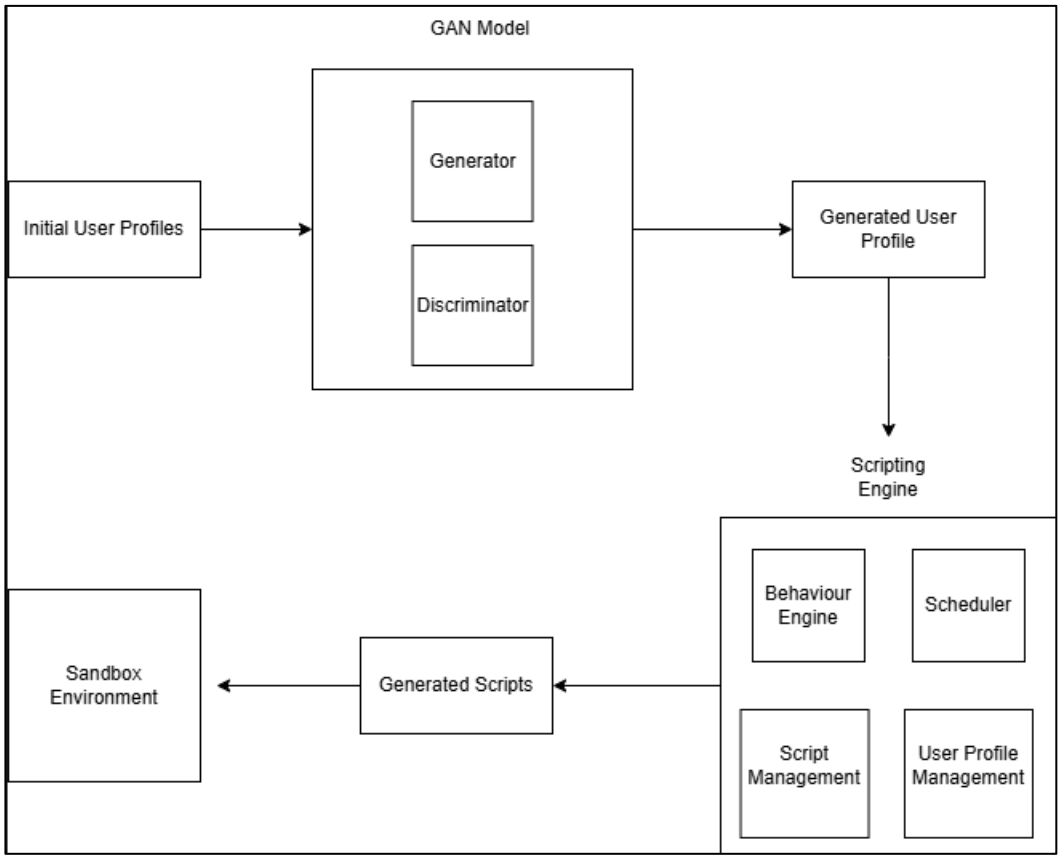


Figure 3: System architecture

- **Initial user profiles**: - First user profiles help to start the process. Derived from publicly accessible databases, these profiles are structured depictions of actual user behavior. Keystroke patterns, digraphs and trigraphs as well as mouse movement metrics are among the features obtained.

- **GAN model**: - The user profiles are sent into a Generative Adversarial Network (GAN). Next neural network components Generator and Discriminator define the GAN. While the Discriminator determines if the user profiles are synthetic or real, the Generator tries to create reasonable synthetic data. By the adversarial training program, the Generator gains knowledge to produce user profiles quite similar to real behavioral patterns.

- **Generated user profile**: - The GAN produces a synthetic yet realistic user behaviour profile. Every profile comprises low-level behavioral information including mouse motion variability, digraph/trigraph sequences, and key stroke dynamics. Search terms for browser behaviour and URLs are created with OpenAI, based on randomly chosen user interests, significantly enrich this profile.

- **Scripting engine**: - The scripting engine is responsible for converting the generated user profile into executable behaviour scripts. It is composed of five key modules:
    - **User Profile Management**: Handles the integration of profile attributes and manages the data used by the engine.
    - **Behaviour Engine**: Updates predefined user activity templates based on profile characteristics.
    - **Script Management**: Selects and builds automation scripts from the updated templates.
    - **Scheduler**: Generates a realistic timetable, distributing activities over time to simulate natural user interaction.

- **Master Automation Logic (internally coordinated)**: Ensures synchronisation and proper execution order of all components.

- **Sandbox environment**: - The final phase of the architecture involves injecting the generated scripts into a virtual environment Cuckoo Sandbox. The scripts are executed within the sandbox, simulating realistic user behaviour such as mouse movements and keyboard input. This activity helps to deceive sandbox-evasive malware into executing, allowing analysts to capture its true behaviour for detection and analysis.

2.3 Data Acquisition and Preprocessing

The development of realistic user profiles relies on the acquisition of behavioural data that accurately reflects natural human interaction patterns. This section outlines the types of data used, the preprocessing steps applied, and the construction of initial user profiles that serve as input to the generative model.

2.3.1 Data Sources

For this research, publicly available datasets were used to obtain user interaction data. The data specifically includes:

- Mouse movement data – capturing cursor trajectories, pauses, and speed variations.
- Keystroke data – including keypress durations, time intervals between key events, and sequence patterns (digraphs and trigraphs).

These data types were selected because they represent low-level behavioural traits that are difficult for malware to distinguish from real user actions when simulated properly.

2.3.2 Construction of Initial User Schema

In order to generate consistent and meaningful user profiles, a structured schema was designed to represent the key behavioural attributes extracted from the preprocessed dataset. This schema serves as a blueprint for how user interactions are modelled and later used by the generative model. It ensures that each user profile maintains a standardized format, allowing the GAN to learn and replicate behaviours more effectively.

The schema is built around five main categories of user behaviour: mouse movement, mouse hovering and interaction, keystroke and typing patterns, word-level structure, and digraph/trigraph dynamics. These categories reflect the types of low-level behaviours that are commonly used by sandbox-evasive malware to detect artificial environments.

Each field in the schema is directly derived from measurable attributes present in the dataset. A brief description of the included categories is as follows:

- **Mouse Movement Metrics**: - This category captures the characteristics of mouse activity, including average and variance of movement speed, distance travelled, and scroll behaviour. Features such as path curvature and direction changes are included to reflect complex, human-like cursor movement patterns.

- **Mouse hover and interaction**: - This covers measures of the mouse's interactions with on-screen components. Simulating user attention and responsiveness using hover time and click frequency helps to replicate user response, therefore adding to the realism of simulated behavior.

- **Typing patterns and keystroke**: - This set of characteristics simulates user typing including speed, corrections, key use including modifiers, and typing

pauses. These behavioral features are essential for spotting patterns that show up organically in human input but are often lacking in synthetic simulations.

- **Text and Word Structures**: - Typing habits can be revealed by word-level statistics including word length, variety, and the ratio of capitalized or lengthy words. These elements set apart structured from informal input patterns.

- **Digraph and Trigraph Characteristics**: - These traits derive from combinations of important sequences. Measurement of typing behavior depends critically on digraphs (two-key combinations) and trigraphs (three-key combinations). Strong behavioural indications are the frequency and presence of common combinations.


```
{
    "mouse_movement_metrics": {
        "avg_mouse_speed_px_s": "Average speed of mouse movement in pixels per second.",
        "var_mouse_speed_px_s": "Variance in the speed of mouse movements.",
        "total_mouse_distance_px": "Total distance the mouse travelled during a session.",
        "avg_distance_per_movement_px": "Average distance covered in a single mouse movement.",
        "avg_path_curvature": "Average curvature of the mouse path.",
        "var_path_curvature": "Variance in the curvature of the mouse path.",
        "diagonal_movements_percent": "Percentage of movements that are diagonal.",
        "mouse_stopping_events": "Number of times the mouse comes to a complete stop.",
        "scroll_events": "Total number of scroll events triggered by the user.",
        "scroll_direction_changes": "Number of times the user changed scrolling direction.",
        "avg_scroll_length_px": "Average scroll length in pixels per event.",
        "frequency_abrupt_stops_mouse": "Frequency of sudden stops in mouse movement."
    },
    "mouse_hover_interaction": {
        "avg_hover_time_ms": "Average time the mouse hovers over an element, in milliseconds.",
        "var_hover_time_ms": "Variance in hover duration.",
        "single_click_frequency": "Frequency of single mouse clicks.",
        "double_click_frequency": "Frequency of double mouse clicks.",
        "right_click_frequency": "Frequency of right-click actions."
    },
    "keystroke_typing_patterns": {
        "typing_speed_wpm": "Typing speed measured in words per minute.",
        "corrections_per_100_keys": "Number of corrections made per 100 keystrokes.",
        "pauses_over_2s": "Number of typing pauses lasting over 2 seconds.",
        "frequency_repeated_text_patterns": "Frequency of repeated text sequences.",
        "frequency_spacebar_enter_usage": "Frequency of using spacebar and enter keys.",
        "caps_lock_usage": "Number of times the Caps Lock key was used.",
        "punctuation_usage": "Number of punctuation marks used.",
        "keyboard_shortcuts_usage": "Count of keyboard shortcut usages (e.g., Ctrl+C)."
    },
    "word_text_structure": {
        "avg_word_length_chars": "Average word length in characters.",
        "variability_word_length_chars": "Variability in word lengths across the session.",
        "short_words_percent": "Percentage of short words typed.",
        "medium_words_percent": "Percentage of medium-length words typed.",
        "long_words_percent": "Percentage of long words typed.",
        "capitalized_words_percent": "Percentage of words that were capitalised."
    },
    "digraph_trigraph_features": {
        "total_digraphs": "Total number of digraphs (two-key combinations) typed.",
        "common_digraphs_percent": "Percentage of digraphs that are commonly used.",
        "total_trigraphs": "Total number of trigraphs (three-key combinations) typed.",
        "common_trigraphs_percent": "Percentage of trigraphs that are commonly used."
    }
}
```


Figure 4: User profile schema

This schema forms the foundation for all user profiles used in this research. Each preprocessed data record is mapped to this schema to create a structured initial user profile. These profiles are then used as input to the Generative Adversarial Network (GAN), allowing for the generation of synthetic user behaviour that mimics real-world patterns.

2.4 User Profile Generation using GAN

2.4.1 Objective of the GAN-Based Generation Approach

The goal of this phase is to create synthetic user behaviour profiles that are similar to the interaction features we see in legitimate human input. A behaviour profile includes the same low-level features that a sandbox-evasive malware might collect to measure an execution environment's authenticity, including the timing of keystrokes and characteristics of mouse movements.

In the past, behaviour simulation methods typically relied on scripted, deterministic behaviours that did not account for the randomness and variability that are foundational aspects of human behaviours. The present study demonstrates a method employing a data-driven approach to learn and recreate behaviour distributions from the genuine data of real users using a Generative Adversarial Network (GAN). The final result is a collection of dynamic user profiles with high-fidelity behaviours that can be used to produce credible user behaviours within the sandbox.

2.4.2 Normalization and Feature Vector Structure

Before training the GAN model, all features extracted from the initial user schema (see Section 2.3.2) are preprocessed and normalised to a range between -1 and 1. This standardisation is critical for stabilizing GAN training and ensuring all feature dimensions contribute equally during learning.

The structure of the input data used to train the GAN is based on a fixed-length feature vector, where each element corresponds to a specific metric from the schema. These include values such as:

- Average mouse speed
- Variance in hover time
- Typing speed
- Frequency of digraph and trigraph sequences
- Use of keyboard shortcuts and corrections

Each feature vector represents a complete user session. The GAN is trained to reproduce these distributions without memorizing specific examples.

2.4.3 WGAN-GP Architecture and Design Rationale

To improve training stability and output quality, the system is built using a Wasserstein GAN with Gradient Penalty (WGAN-GP). This framework addresses common GAN training issues such as mode collapse and vanishing gradients.

Generator Architecture

The Generator transforms a 100-dimensional latent vector (sampled from a standard normal distribution) into a high-dimensional user behaviour profile. The network uses multiple fully connected layers with Batch Normalization and Leaky ReLU activations to stabilize gradient flow and accelerate convergence.

Layer Structure:

- Linear(100 → 256) → BatchNorm → LeakyReLU
- Linear(256 → 512) → BatchNorm → LeakyReLU
- Linear(512 → 1024) → BatchNorm → LeakyReLU
- Linear(1024 → N) → Tanh

(Where N = number of features in the user schema)

Critic Architecture

The Critic (Discriminator) evaluates whether the input profile is real or synthetic. It uses Layer Normalisation (instead of BatchNorm) to avoid feedback instability and includes Dropout layers to improve robustness.

Layer Structure:

- Linear(N → 1024) → LayerNorm → LeakyReLU → Dropout
- Linear(1024 → 512) → LayerNorm → LeakyReLU → Dropout
- Linear(512 → 256) → LayerNorm → LeakyReLU
- Linear(256 → 1)

Unlike traditional GANs, no sigmoid activation is used in the output. Instead, the model directly estimates the Wasserstein distance between real and fake distributions.

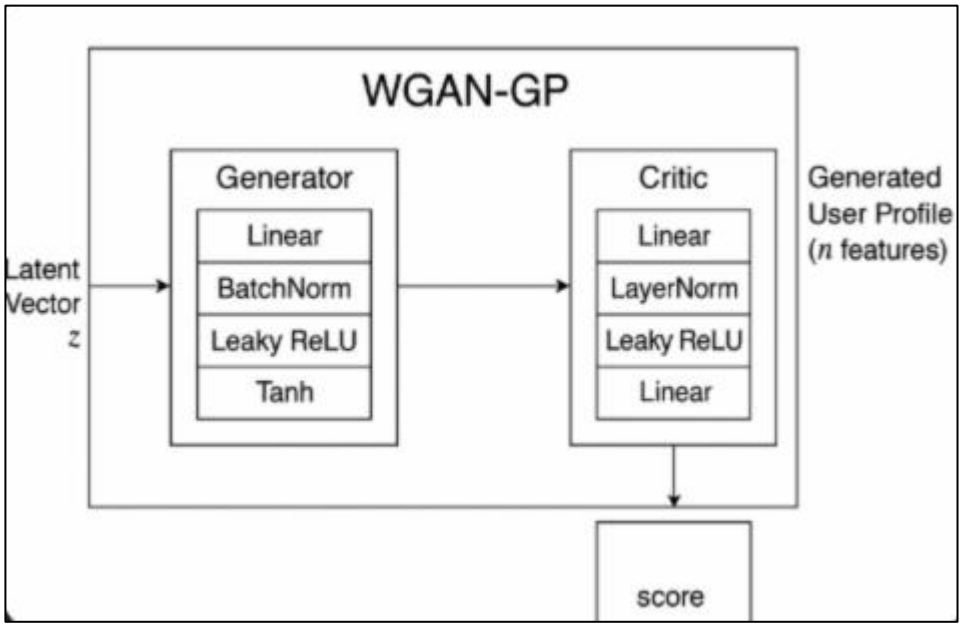


Figure 5: WGAN-GP architecture

2.4.4 Training Configuration and Stability Enhancements

Training Generative Adversarial Networks (GANs) is notoriously difficult due to issues such as mode collapse, unstable gradient updates, and non-convergent loss behaviour. To address these challenges, the user behaviour generation model in this study adopts the Wasserstein GAN with Gradient Penalty (WGAN-GP) architecture,

paired with several advanced training strategies and custom regularisation techniques to improve both training stability and the quality of generated profiles.

Learning Rate and Optimizers

Both the Generator and Critic networks are optimized using the Adam optimizer, with parameters specifically chosen to improve convergence:

- Learning rate: 0.0001
- $\beta 1 = 0.5$, $\beta 2 = 0.9$ (standard for WGAN stability)

To prevent premature saturation or vanishing gradients, learning rate schedulers are used for both networks. These schedulers reduce the learning rate at specific training milestones (e.g., epochs 500, 1000, and 1500), gradually slowing down learning as convergence is approached.

Training Configuration Parameters

Key hyperparameters used in training include:

- Latent dimension: 100 (size of the noise input vector to the Generator)
- Batch size: 64
- Number of epochs: up to 2000
- n_critic: 5 (the Critic is updated 5 times for every Generator update)
- Gradient penalty λ: 10 (coefficient used in regularising the Critic)

Gradient Penalty for Lipschitz Continuity

A key component of WGAN-GP is the gradient penalty, which replaces weight clipping from the original WGAN. This penalty enforces the Lipschitz constraint on the Critic by encouraging gradients to remain close to 1 during interpolation between real and fake samples.

Early Stopping Strategy

To avoid overfitting and resource waste, an early stopping mechanism is used. The training loop continuously evaluates the Generator's output based on statistical similarity to real data. If no improvement is observed over a fixed number of evaluation intervals (e.g., 200 epochs), training is halted automatically, and the best Generator state is saved.

This prevents degradation in output quality, which is a common issue in GANs trained for extended periods.

2.4.5 Synthetic Profile Generation and Postprocessing

Once the Generator has been trained, it can be used to produce synthetic user profiles by sampling random noise vectors from a latent space. These vectors are passed through the Generator network, which transforms them into feature vectors that resemble real user behaviour.

Each output vector contains a complete set of behavioural attributes such as average mouse speed, typing speed, hover duration, and digraph frequency structured according to the user schema defined earlier. These vectors are still in their normalized form, scaled between -1 and 1, as required by the training process.

To make the synthetic profiles usable by downstream components like the scripting engine, the generated values must be transformed back to their original scale. This process, known as denormalization, uses the stored minimum and maximum values from the original dataset to reverse the scaling. The result is a profile that reflects human-like behaviour patterns in real-world units (e.g., characters per second, pixels per second, milliseconds, etc.).

Distribution Matching

Even after denormalization, there may still be minor differences between the statistical distributions of real and synthetic data. In order to add a layer of realism, a distribution

matching method has been added as a postprocessing step that alters the synthetic data by matching distributional properties to that of the real dataset as a postprocessing step.

The synthetic values for each feature are adjusted such that their distribution matches that of the real values for the corresponding feature using a percentile-matching strategy. Therefore, the width of synthetic feature values retains statistical shape compatibility even if the GAN outputs slightly drift. This postprocessing step improves the realism of generated data by correcting small deviations without impacting the relationships or behaviour of the data patterns.

A visual representation of the generation and postprocessing workflow is shown in Figure 6.

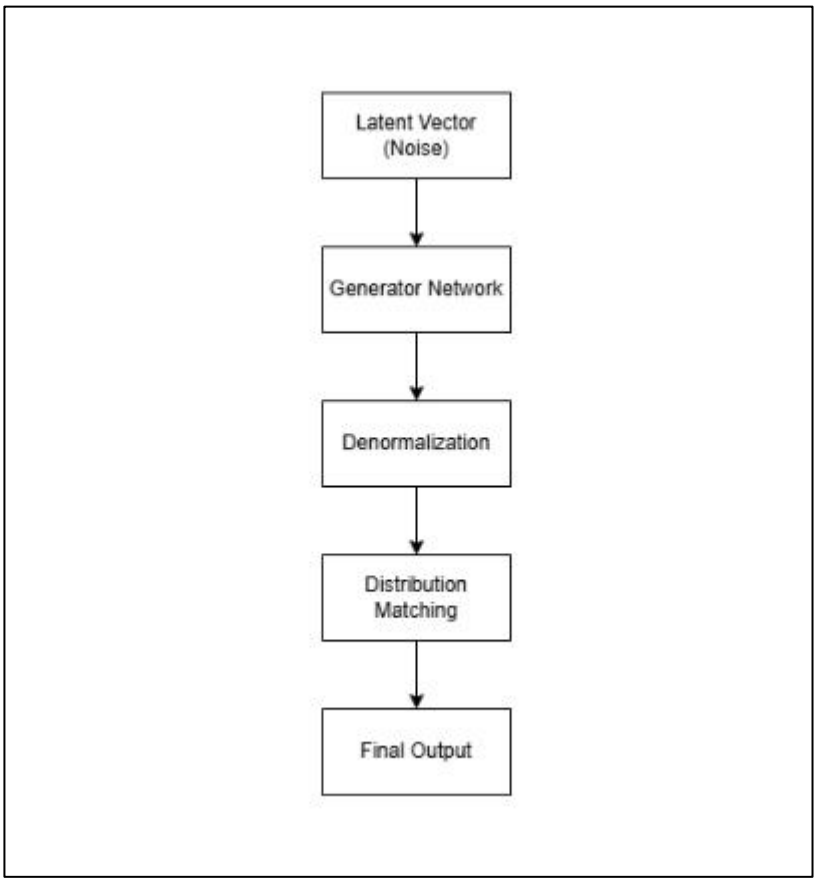


Figure 6: Generation and postprocessing workflow

- **Latent Vector (Noise)** – Random input sampled from a normal distribution
- **Generator Network** – Produces a normalised behavioural profile
- **Denormalisation** – Converts the profile into real-world values
- **Distribution Matching** – Aligns the output with real data distributions
- **Final Output** – A complete, human-like user behaviour profile

Below figure 7 shows a generated user profile using the WGAN-GP model.


```
🔍 Generated User Profile:
[
    {
        "avg_mouse_speed_px_s":373.4039066505,
        "var_mouse_speed_px_s":55.240733552,
        "total_mouse_distance_px":3592.4276051402,
        "avg_distance_per_movement_px":224.0663007203,
        "avg_path_curvature":0.4781393528,
        "var_path_curvature":0.332795608,
        "avg_hover_time_ms":1086.7153979743,
        "var_hover_time_ms":233.8082518566,
        "avg_scroll_length_px":251.2282219198,
        "diagonal_movements_percent":40.8620772552,
        "avg_word_length_chars":6.7604810506,
        "variability_word_length_chars":2.2121785879,
        "short_words_percent":37.620656569,
        "medium_words_percent":55.5363440514,
        "long_words_percent":6.1098894477,
        "capitalized_words_percent":9.178208828,
        "scroll_percentage":70.4458377844,
        "focused_browsing_percent":62.9373497233,
        "single_click_frequency":49.3219870329,
        "double_click_frequency":5.8581960201,
        "right_click_frequency":5.8352919817,
        "scroll_events":21.3023298979,
        "frequency_abrupt_stops_mouse":14.4844514132,
        "typing_speed_wpm":29.0968880057,
        "corrections_per_100_keys":11.2836056948,
        "pauses_over_2s":13.5629764199,
        "frequency_repeated_text_patterns":6.121175766,
        "frequency_spacebar_enter_usage":14.4753834605,
        "caps_lock_usage":1.307477206,
        "punctuation_usage":14.0154078603,
        "total_digraphs":139.6879136562,
        "common_digraphs_percent":27.2695434093,
        "total_trigraphs":173.7811312079,
        "common_trigraphs_percent":13.3035737276,
        "words_typed_per_session":475.0028371811,
        "mouse_stopping_events":12.8840005398,
        "scroll_direction_changes":8.2555548847,
        "keyboard_shortcuts_usage":8.5014643669
    }
]
```


Figure 7: WGAN-GP model generated output

2.4.6 Integration with OpenAI for Profile Enrichment

The GAN model does not consider contextual factors of user activity, including browsing behavior or interest-driven activities, even if it efficiently creates the behavioral traits of a user (such as keyboard patterns and mouse dynamics). The solution uses OpenAI's language model APIs in an enrichment layer to replicate reasonable user context depending on specified interests, hence addressing this.

Interest Selection

From a pre-defined, large pool of categories e.g., technology, fitness, travel, finance each synthetic user is assigned a set of four random interests. These interests allow one to replicate the variety and precision of actual user intentions.

OpenAI Integration

The system uses OpenAI to do two enrichment chores for every given interest:

- **Search Term Generation** – Realistic queries a user with that interest might search for online.
- **URL Generation** – Plausible websites a user might visit, consistent with their interest-driven behaviour.

This integration gives otherwise numerical user profiles semantic and activity context therefore enabling downstream simulation scripts to replicate more complete human activities.

```
Generated Prompt:

You are an expert in internet trends and user behavior. For the following interests:
- Board games, Acrylic painting, Aerobics, Collecting antiques,
generate:
1. Four realistic search terms for each interest.
2. Four popular websites for each interest.
Also, include one trending topic globally and generate:
1. Four search terms for this trending topic.
2. Four popular websites for this trending topic.

Response format (strictly adhere to this structure):
{
    "interests": [
        {
            "interest": "<interest name>",
            "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
            "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
        },
        ...
    ],
    "trending_topic": {
        "topic": "<trending topic>",
        "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
        "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
    }
}
```

Figure 8: Generated OpenAI prompt

```
{
   "interests": [
       {
           "interest": "Board games",
           "search_terms": ["best new board games", "classic board games", "board game reviews", "how to play chess"],
           "websites": ["www.boardgamegeek.com", "www.theguardian.com/games/series/board-games", "www.games-workshop.com", "www.zmangames.com"]
       },
       {
           "interest": "Acrylic painting",
           "search_terms": ["acrylic painting techniques", "how to start acrylic painting", "landscape acrylic painting", "best acrylic paints"],
           "websites": ["www.artsy.net", "www.jerrysartarama.com", "www.liquitex.com", "www.goldenpaints.com"]
       },
       {
           "interest": "Aerobics",
           "search_terms": ["aerobics workout for weight loss", "beginner aerobics steps", "best aerobics classes", "Zumba workouts"],
           "websites": ["www.healthline.com", "www.self.com", "www.shape.com", "www.myfitnesspal.com"]
       },
       {
           "interest": "Collecting antiques",
           "search_terms": ["how to value antiques", "popular antique items", "antique shows near me", "restoring antiques"],
           "websites": ["www.antiquesroadshow.com", "www.kovels.com", "www.the-saleroom.com", "www.barnebys.com"]
       }
   ],
   "trending_topic": {
       "topic": "Metaverse",
       "search_terms": ["Metaverse explained", "leading companies in metaverse", "how to experience the metaverse", "real estate in metaverse"],
       "websites": ["www.techcrunch.com", "www.medium.com/tag/metaverse", "www.theguardian.com/technology/metaverse", "www.futureofthemetaverse.com"]
   }
}
```


Figure 9: OpenAI response with search terms & URLs

2.4.7 Final Output Profile Format

The final output of the synthetic profile generation pipeline is a fully enriched user profile that combines:

- Behavioural features generated by the WGAN-GP model
- Interest-based enrichment data (search terms and visiting URLs) obtained via OpenAI

This profile is designed to reflect both the low-level interaction patterns (e.g., keystrokes, mouse dynamics) and high-level interest-driven behaviour, enhancing the realism of simulated users in sandbox environments.

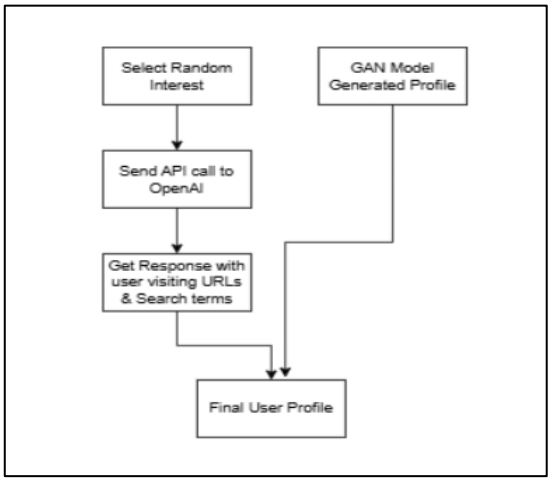


Figure 10: Complete user profile creation workflow

Below figure 11, shows an enhanced final user profile.

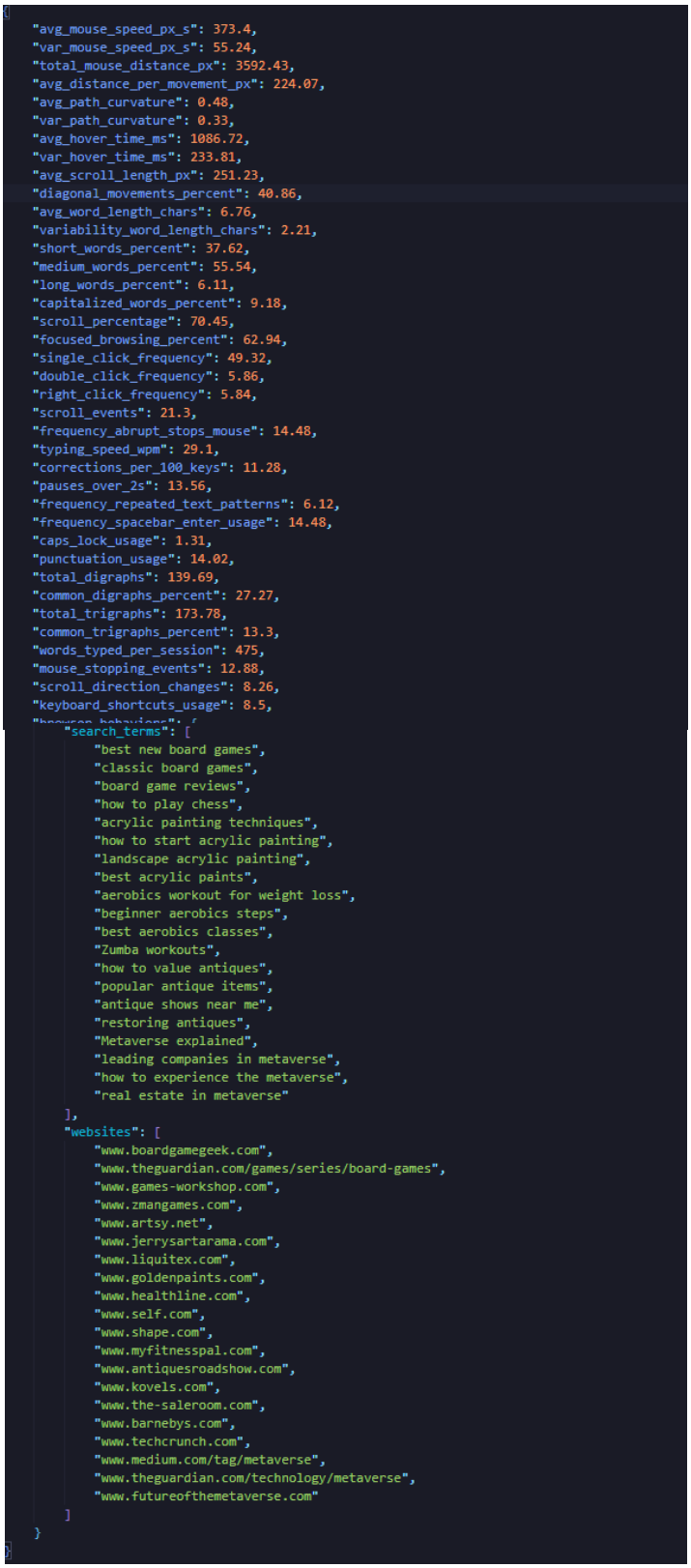


Figure 11: Complete user profile

2.5 Scripting Engine

2.5.1 Overview of the Scripting Engine Module

The Scripting Engine Module is a fundamental part of the system that takes the generated user profiles and turns them into executable behaviour scripts. Its goal is to imitate real user activity in a sandboxed environment, making it seem as if the malware is activated as if a real user was conducting a real session.

This module acts as the bridge between the output of the user profile generation pipeline and the final execution phase inside the sandbox. It takes enriched user profiles containing both behavioural attributes and interest-based metadata and dynamically generates a time-structured script that reflects genuine user activity.

To support a flexible and scalable simulation system, the scripting engine is designed using a modular architecture. It comprises five specialised submodules:

- **Master Automation Module** – Orchestrates and synchronizes the execution of all submodules in the correct order.
- **User Profile Management Module** – Handles the parsing, validation, and distribution of attributes from the input user profile.
- **Behaviour Engine Module** – Updates predefined behaviour templates using parameters from the user profile (e.g., typing speed, scroll activity).
- **Script Management Module** – Organizes, selects, and compiles appropriate scripts based on the updated templates and simulation context.
- **Scheduler and Executor Module** – Generates the timetable, assigns activities over time, and executes the full script within the sandbox.

The scripting engine ensures that each simulation is context-aware, personalized, and non-repetitive, which is critical for deceiving sandbox-aware malware. It supports a mix of both static and dynamic activity scripts, blending system-level artefact creation with user-like interaction patterns.

This architecture promotes extensibility, enabling the addition of new script templates or behaviour types without modifying the core pipeline. The following sections describe each of these modules in detail.

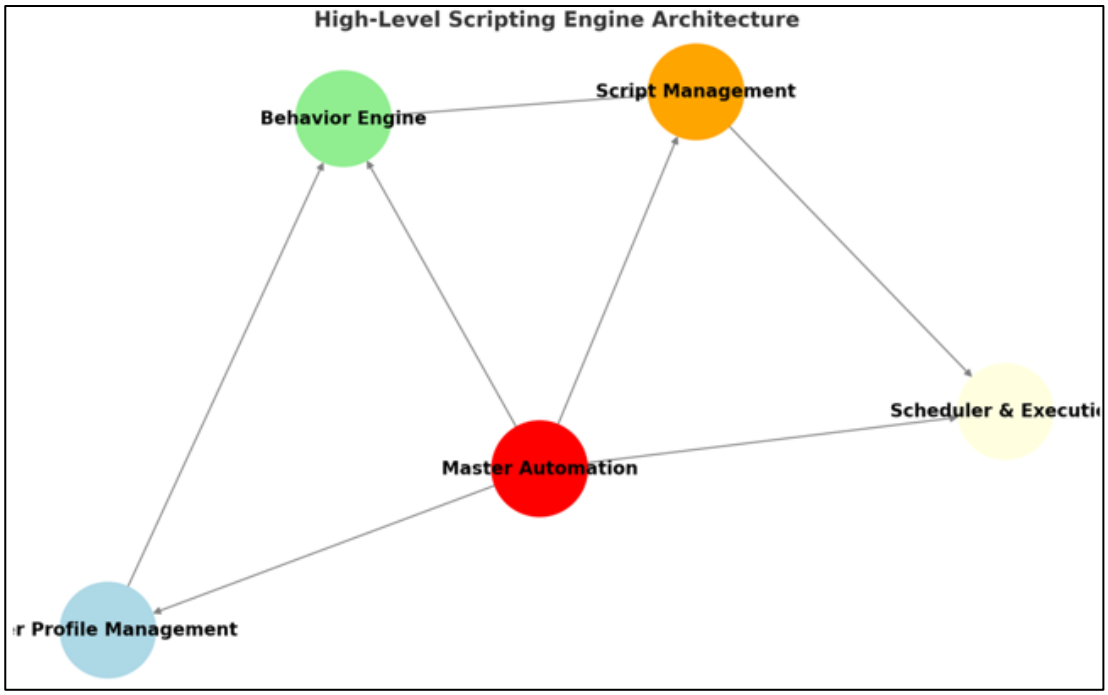


Figure 12: Scripting engine architecture

2.5.2 Master Automation Module

The Master Automation Module is the central controller of the Scripting Engine. Its role is to coordinate and manage the execution flow of all subordinate modules, ensuring that each step in the script generation pipeline is performed in the correct order and with the required input data.

This module does not perform any user behaviour logic itself, but acts as the automation orchestrator, supervising:

- The handover of enriched user profiles to the User Profile Management Module.
- The use of extracted parameters by the Behaviour Engine Module to personalise script templates.

- The selection and compilation of scripts by the Script Management Module.
- The generation of realistic schedules and triggering of simulation by the Scheduler and Executor Module.

By maintaining a consistent execution sequence and resolving inter-module dependencies, the Master Automation Module ensures the scripting engine runs as a self-contained, autonomous system. It also handles status monitoring, logging, and error handling during the process, which improves the stability and debuggability of the entire framework.

This modular approach allows each subcomponent to remain independent and reusable, while the Master Automation Module ensures that they work together seamlessly to produce a complete and coherent behaviour simulation package.

2.5.3 User Profile Management Module

The scripting engine pipeline starts with the User Profile Management Module. Its main duties are parse, validate, and extract pertinent attributes from the enhanced user profiles produced in the past phases of the system.

Every user profile is presented in a structured JSON style and combines:

- Low-level behavioral measures (digraph/trigraph frequencies, mouse movement patterns, typing speed, etc.)
- Enrichment motivated by interest (e.g., particular interests, search keywords, visiting URLs)

The module does the following main purposes:

Profile validation and parsing

It checks that all expected fields follow the preset schema after reading the arriving JSON profile. This covers searches for missing elements, erroneous data types, or invalid ranges. Failed validation profiles are either deleted or marked for inspection to

prevent creating unrealistic or flawed scripts.

Attribute Extraction

Once approved, the module gathers particular qualities needed for the downstream modules. These values are dynamically changed in the Behaviour Engine Module by use of a structured framework.

Examples of extracted attributes include:

- Mean typing speed and key hold durations
- Mouse movement speed and direction change frequency
- Time-based interaction pacing
- List of search terms and URLs associated with the user's interest

Routing Profile

The processed profile passes to the Behaviour Engine and Scheduler modules upon extraction. This enables the simulation system to synchronize what actions are carried out (based on behavioural characteristics) with when they are carried out (based on scheduling logic).

2.5.4 Behaviour Engine Module

Generating generalised user behaviour patterns based on the enhanced user profile falls to the Behaviour Engine Module. Among these behaviors are mouse movement dynamics, typing qualities, and scrolling action. The Behaviour Engine generates reusable interaction segments reflecting the natural input style of a given user instead of whole scripts.

Ensuring behavioral consistency throughout a simulation session depends mostly on this module. Whether the user is browsing, editing a document, or engaging with system settings any activity the user's interaction characteristics (e.g., typing speed, mouse jitter, scroll depth) stay consistent and realistic.

Behaviour Pattern Generation

The module begins by analysing specific attributes extracted from the user profile, such as:

- Average and variance of keypress duration and inter-key time
- Mouse speed, path curvature, and direction changes
- Frequency and duration of scrolling activity
- Hover time and click distributions

Using these values, the Behaviour Engine produces behavioural components such as:

- Mouse movement sequences that simulate natural cursor paths
- Typing patterns including character-by-character delays and pauses
- Scrolling actions with variable lengths, pauses, and velocity

These actions have nothing to do with any one activity. Rather, they create a library of interaction behaviours that can be reused among several script templates, hence encouraging realism and efficiency.

2.5.5 Script Management Module

Managing, choosing, and compiling the predefined Python-based user behavior scripts needed to replicate reasonable desktop activity falls to the Script Management Module. These models are meant to show typical user behavior in several spheres, including online surfing, file editing, program running, system interactions, multimedia handling, and so on.

This module addresses activity-specific scripting logic unlike the Behavourable Engine, which creates abstract behavioural patterns including typing, scrolling, and mouse movement. Every script template is parameterised and meant to be changed with the user-specific behaviour produced later in the pipeline.

2.5.6 Scheduler and Executor Module

The Scheduler and Executor Module is the final component of the scripting engine. Its role is to determine when each user activity should take place during the simulation session and to ensure that the final set of scripts is executed in a timed and controlled manner within the sandbox environment.

This module uses both static time structuring and profile-driven variability to simulate realistic user engagement patterns. It consists of two tightly integrated components: the Timetable Engine and the Execution Manager.

Timetable Engine Function

Producing a realistic activity schedule for the simulation session falls to the timetables engine. The beginning time, length, and order of every chosen activity are specified in this calendar.

Key factors considered during timetable generation include:

- The duration of each script template (as defined in the script metadata)
- The total session length (configured per experiment)
- The need to include natural gaps or idle periods between actions
- Variation in activity order across different users to reduce predictability

The generated timetable is output as a structured file (typically JSON or CSV) that serves as a map for when and how scripts should be executed.

Selection of Behaviours Based on Schedule

Once the timetable is generated, the module maps each time slot to a specific activity:

- Activities are selected based on duration fit, type diversity, and remaining session time.
- The script templates corresponding to each activity are already personalized (via the Script Management Module) and ready for execution.

- Optional randomisation logic may be applied to avoid deterministic behaviour ordering.

This process ensures that the simulation timeline exhibits natural pacing, with varying intensity, periods of inactivity, and a mix of short and long interactions closely resembling real user workflows.

Figure 13 shows a generated timetable by the scheduler and executor module.

```
[
    {
        "execution_time_min": 0,
        "behavior_id": 9,
        "script": "switching_active_window.py",
        "duration_min": 3,
        "notes": "Switch active window"
    },
    {
        "execution_time_min": 3,
        "behavior_id": 5,
        "script": "create_modify_text_files.py",
        "duration_min": 4,
        "notes": "Copy files, create folders and save them"
    },
    {
        "execution_time_min": 7,
        "behavior_id": 17,
        "script": "fake_usb_interactions.ps1",
        "duration_min": 3,
        "notes": "USB interactions"
    },
    {
        "execution_time_min": 10,
        "behavior_id": 16,
        "script": "multiple_software_interactions.py",
        "duration_min": 5,
        "notes": "Multiple software interactions"
    },
    {
        "execution_time_min": 15,
        "behavior_id": 4,
        "script": "create_modify_text_files.py",
        "duration_min": 10,
        "notes": "File Creation and Modification in Notepad"
    },
    {
        "execution_time_min": 25,
        "behavior_id": 15,
        "script": "view_video.py",
        "duration_min": 5,
        "notes": "View video"
    },
    {
        "execution_time_min": 30,
        "behavior_id": 20,
        "script": "windows_file_search.py",
        "duration_min": 4,
        "notes": "Search files in Explorer"
    },
    {
        "execution_time_min": 34,
        "behavior_id": 15,
        "script": "view_video.py",
        "duration_min": 5,
        "notes": "View video"
    },
```


Figure 13: Generated time table

2.5.7 Scripting Engine Workflow

This subsection describes the overall workflow that the scripting engine follows once a user profile is generated. The workflow documents how the process is automated behavioural data from users into a scheduled simulation within the sandbox.

The process begins with extracting appropriate behavioural traits (e.g., typing speed, mouse jitter, scrolling intervals, etc.) from the enriched user profile. These parameters are used to generate a realistic set of baseline behaviours. Activity script templates are chosen and updated with behaviour to ensure these generated actions are consistent with real user behaviours. Finally, the scheduling engine generates an activity schedule for each of the scripts. After time mapping the scripts, these scripts are packaged and loaded into the sandbox.

Figure 14 shows a flow diagram to illustrate this sequence of steps:

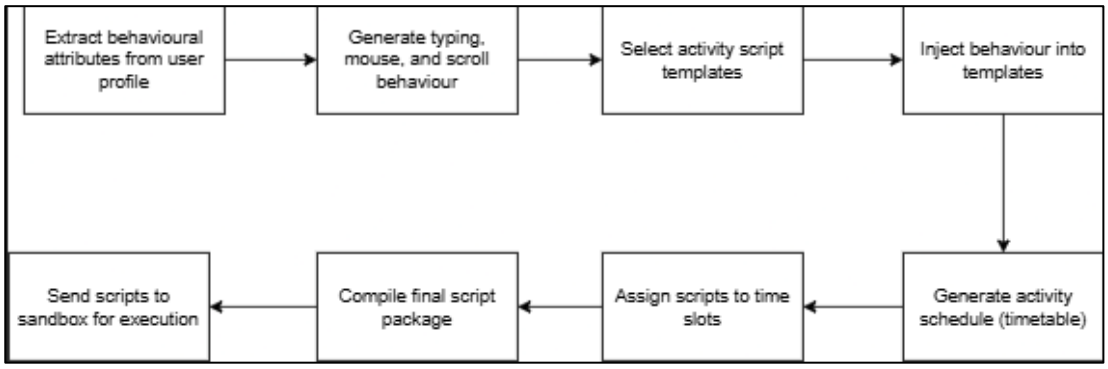


Figure 14: Scripting engine workflow

2.6 Sandbox Integration

Once the full set of personalised behaviour scripts and the corresponding timetable have been generated, the final simulation package is sent to the sandbox environment. The sandbox is the execution environment where the synthetic user behaviour is replayed in real time, with the goal of triggering and observing the actions of sandbox-evasive malware.

This research uses Cuckoo Sandbox, a well-established, open-source malware analysis platform that supports virtualised environments and can be extended to support user interaction automation.

2.7 Implementation & Testing

This chapter presents the practical implementation of the proposed system and outlines the testing conducted to ensure its correctness and functionality. The system is composed of several interdependent modules, including a generative model for user profile synthesis, a scripting engine for behaviour simulation, and a sandbox integration layer for malware analysis.

2.7.1 WGAN-GP Implementation

Data Preparation and Normalization

This section handles the initial setup and preparation of behavioural data for training the GAN model. First, the system allows the user to upload a CSV file containing real user behaviour data (such as keystrokes and mouse movements). Once uploaded, the dataset is loaded and basic information is printed for inspection.

To ensure stable training of the neural networks, the raw data is normalised to a standard range between -1 and 1. This transformation helps the model learn more efficiently. Additionally, the minimum and maximum values of each feature are stored so that the generated synthetic data can later be converted back (denormalised) to realistic, original scales.

Finally, the prepared data is converted into PyTorch tensors and made ready for input to the model.

```python
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, TensorDataset
from scipy import stats
from google.colab import files
import time
from tqdm import tqdm

# Upload file manually
print("Please upload your CSV file with behavioral biometrics data")
uploaded = files.upload()
filename = list(uploaded.keys())[0]  # Get uploaded file name
print(f"Using uploaded file: {filename}")

# Check if running on GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load dataset
df = pd.read_csv(filename)
print(f"Dataset shape: {df.shape}")
print("First few rows of the dataset:")
print(df.head())

# Normalize dataset between -1 and 1 for stability
def normalize_data(df):
    return 2 * ((df - df.min()) / (df.max() - df.min())) - 1

# Denormalize data back to original range
def denormalize_data(df, ref_df):
    return ((df + 1) / 2) * (ref_df.max() - ref_df.min()) + ref_df.min()

# Store original min/max for later denormalization
df_min = df.min()
df_max = df.max()

# Normalize data
df_scaled = normalize_data(df)

# Convert dataset to tensors
real_data = torch.tensor(df_scaled.values, dtype=torch.float32).to(device)
feature_dim = real_data.shape[1]  # Number of features
```

Figure 15: Data preparation & normalization implementation

GAN Training Hyperparameters

This section defines the critical hyperparameters used to control the training of the WGAN-GP model.

```python
# WGAN-GP Hyperparameters
latent_dim = 100  # Size of random noise vector
batch_size = 64
n_critic = 5  # Number of critic updates per generator update
lambda_gp = 10  # Gradient penalty coefficient
epochs = 2000
early_stopping_patience = 200
eval_interval = 50
lr = 0.0001  # Learning rate for both networks
beta1 = 0.5
beta2 = 0.9
```

Figure 16: WGAN-GP hyperparameter initialization

Generator and Critic Network Implementation

This sub-section defines the core components of the WGAN-GP architecture: the Generator and the Critic (Discriminator).

The Generator takes a latent vector (random noise) and transforms it into a synthetic user profile.

The Critic distinguishes between real and synthetic samples by estimating their Wasserstein distance.

```python
# Define Generator with Batch Normalization
class Generator(nn.Module):
    def __init__(self, latent_dim, feature_dim):
        super(Generator, self).__init__()
        self.latent_dim = latent_dim
        self.feature_dim = feature_dim

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),

            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),

            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),

            nn.Linear(1024, feature_dim),
            nn.Tanh()  # Output in range [-1, 1]
        )

    def forward(self, z):
        # Ensure z has the right shape for BatchNorm
        if z.dim() == 2:
            return self.model(z)
        else:
            z_reshaped = z.view(-1, self.latent_dim)
            return self.model(z_reshaped)

# Define Critic (Discriminator without Sigmoid)
class Critic(nn.Module):
    def __init__(self, feature_dim):
        super(Critic, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(feature_dim, 1024),
            nn.LayerNorm(1024),  # Layer normalization instead of batch norm
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(1024, 512),
            nn.LayerNorm(512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.LeakyReLU(0.2),

            nn.Linear(256, 1)  # No activation for Wasserstein distance
        )

    def forward(self, x):
        return self.model(x)
```

Figure 17: Generator & critic network implementation

Model Initialization and Optimization Setup

This section outlines the preparation steps before training the GAN model:

- **Model Initialization**: The Generator and Critic models are instantiated and moved to the selected device (CPU or GPU).
- **Optimizers**: Both models use the Adam optimizer with tuned beta parameters.
- **Learning Rate Schedulers**: Learning rates are reduced at specific milestones to promote stable convergence.

```
# Initialize models
generator = Generator(latent_dim, feature_dim).to(device)
critic = Critic(feature_dim).to(device)

# Optimizers
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, beta2))
optimizer_C = optim.Adam(critic.parameters(), lr=lr, betas=(beta1, beta2))
```

Figure 18: Model initialization & optimization setup

Gradient Penalty Implementation

To enforce the Lipschitz constraint required by the WGAN-GP formulation, a gradient penalty is computed.

```
# Compute gradient penalty
def compute_gradient_penalty(critic, real_samples, fake_samples):
    # Random weight for interpolation
    alpha = torch.rand(real_samples.size(0), 1).to(device)

    # Interpolation between real and fake samples
    interpolates = (alpha * real_samples + (1 - alpha) * fake_samples).requires_grad_(True)

    # Get critic output for interpolated samples
    d_interpolates = critic(interpolates)

    # Get gradients with respect to inputs
    fake = torch.ones(real_samples.size(0), 1).to(device)
    gradients = torch.autograd.grad(
        outputs=d_interpolates,
        inputs=interpolates,
        grad_outputs=fake,
        create_graph=True,
        retain_graph=True,
        only_inputs=True,
    )[0]

    # Compute gradient penalty
    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * lambda_gp
    return gradient_penalty
```

Figure 19: Gradient penalty implementation

Correlation Loss Function

To enhance the realism of the synthetic profiles, a correlation-based loss function is introduced. This function compares the pairwise correlations between behavioural features in both real and generated data.

```python
# Feature correlation loss
def correlation_loss(real_batch, fake_batch):
    # Calculate correlation matrices
    real_corr = torch.corrcoef(real_batch.T)
    fake_corr = torch.corrcoef(fake_batch.T)

    # Get upper triangular elements (excluding diagonal)
    mask = torch.triu(torch.ones_like(real_corr), diagonal=1).bool()
    real_corr_triu = real_corr[mask]
    fake_corr_triu = fake_corr[mask]

    # Mean squared error between correlation matrices
    return torch.mean((real_corr_triu - fake_corr_triu) ** 2)

# Generate synthetic data
def generate_synthetic_data(generator, num_samples=1000):
    generator.eval()  # Set to evaluation mode
    with torch.no_grad():
        noise = torch.randn(num_samples, latent_dim).to(device)
        generated_samples = generator(noise).detach().cpu().numpy()
    generator.train()  # Set back to training mode
    return generated_samples

# Evaluate synthetic data using KS test
def evaluate_synthetic_data(real_df, synthetic_df):
    ks_results = {}
    for column in real_df.columns:
        stat, p_value = stats.ks_2samp(real_df[column], synthetic_df[column])
        ks_results[column] = p_value

    # Average p-value across all features
    avg_p_value = sum(ks_results.values()) / len(ks_results)
    return avg_p_value, ks_results
```

Figure 20: Correlation loss function implementation

Training loop

The training loop constitutes the core iterative process of the WGAN-GP model. During each epoch, real user profile batches are sampled and passed through the critic to compute their scores. Simultaneously, the generator produces synthetic profiles from random latent vectors. The critic is trained using the Wasserstein loss function with a gradient penalty to ensure Lipschitz continuity.

Every n_critic steps, the generator is updated to minimize the critic's score on synthetic data while also incorporating a correlation loss component. This correlation loss ensures the preservation of statistical relationships (feature-wise) between real and generated profiles. Both generator and critic are optimized using Adam optimizer with learning rate schedulers. Losses for each step are logged for performance monitoring and future visualization.

```python
# Training Loop
start_time = time.time()
for epoch in range(epochs):
    for i, real_batch in enumerate(dataloader):
        real_samples = real_batch[0].to(device)
        batch_size = real_samples.size(0)

        # ---------------------
        # Train Critic
        # ---------------------
        optimizer_C.zero_grad()

        # Generate fake samples
        z = torch.randn(batch_size, latent_dim).to(device)
        fake_samples = generator(z).detach()

        # Compute critic loss with gradient penalty
        real_validity = critic(real_samples)
        fake_validity = critic(fake_samples)

        # Wasserstein loss
        critic_loss = -torch.mean(real_validity) + torch.mean(fake_validity)

        # Gradient penalty
        gp = compute_gradient_penalty(critic, real_samples, fake_samples)

        # Total critic loss
        critic_loss = critic_loss + gp

        critic_loss.backward()
        optimizer_C.step()

        # Store critic loss
        c_loss_history.append(critic_loss.item())

        # ---------------------
        # Train Generator
        # ---------------------
        # Train generator every n_critic steps
        if i % n_critic == 0:
            optimizer_G.zero_grad()

            # Generate fake samples
            z = torch.randn(batch_size, latent_dim).to(device)
            fake_samples = generator(z)

            # Get critic score
            fake_validity = critic(fake_samples)

            # Add correlation loss
            corr_loss = correlation_loss(real_samples, fake_samples)

            # Generator loss (minimize negative critic score with correlation penalty)
            gen_loss = -torch.mean(fake_validity) + 0.1 * corr_loss

            gen_loss.backward()
            optimizer_G.step()
```

Figure 21: WGAN-GP training loop

2.7.2 User Profile Enrichment with OpenAI

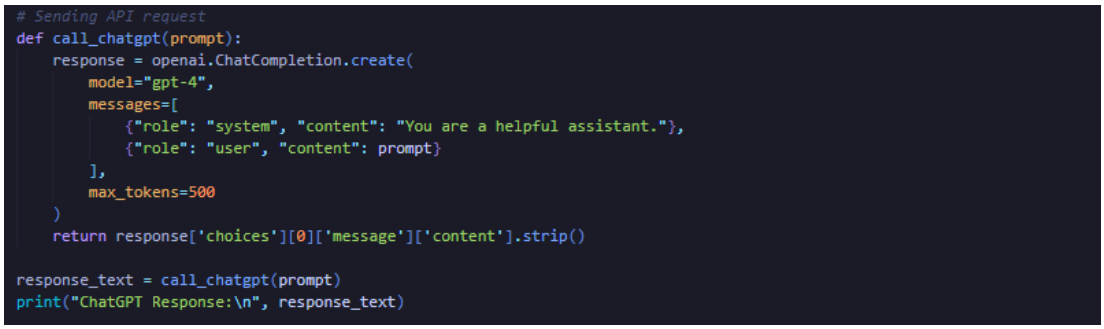
This section describes the steps taken to enrich synthetic user profiles based on realistic web browsing behaviour using OpenAI. First, the GAN model generates a synthetic profile with certain behavioural characteristics. The methodology then selects a random user interests set from a fixed user interest set. This set of interests will then

be collectively used to create a structured prompt, which is sent to OpenAI using the API. For each interest, there are two things the prompt requests:

1. Search terms that are likely realistic for a user to search for on a search engine.
2. Relevant websites or URLs that are likely on topic, and that a user might visit when exploring that topic.

```
import openai

# OpenAI API setup
openai.api_key = [redacted]

# Creating a prompt for OpenAI
def create_prompt(interests):
    prompt = f"""
You are an expert in internet trends and user behavior. For the following interests:
- {', '.join(interests)},
generate:
1. Four realistic search terms for each interest.
2. Four popular websites for each interest.
Also, include one trending topic globally and generate:
1. Four search terms for this trending topic.
2. Four popular websites for this trending topic.

Response format (strictly adhere to this structure):
{{
    "interests": [
        {{
            "interest": "<interest name>",
            "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
            "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
        }},
        ...
    ],
    "trending_topic": {{
        "topic": "<trending topic>",
        "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
        "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
    }}
}}

Example response:
{{
    "interests": [
        {{
            "interest": "Soccer",
            "search_terms": ["latest soccer news", "soccer match highlights", "soccer player stats", "soccer world cup"],
            "websites": ["www.fifa.com", "www.espn.com/soccer", "www.soccerway.com", "www.goal.com"]
        }},
        {{
            "interest": "AI",
            "search_terms": ["artificial intelligence news", "best AI tools 2024", "AI trends", "AI startups"],
            "websites": ["www.openai.com", "www.aimagazine.com", "www.techcrunch.com", "www.wired.com"]
        }}
    ],
    "trending_topic": {{
        "topic": "Electric Vehicles",
        "search_terms": ["latest electric vehicles", "electric vehicle news", "best EVs 2024", "EV charging stations near me"],
        "websites": ["www.tesla.com", "www.electrek.co", "www.insideevs.com", "www.caranddriver.com"]
    }}
}}
    """
    return prompt
```

Figure 22: Prompt generation for OpenAI



```
# Sending API request
def call_chatgpt(prompt):
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ],
        max_tokens=500
    )
    return response['choices'][0]['message']['content'].strip()

response_text = call_chatgpt(prompt)
print("ChatGPT Response:\n", response_text)
```

Figure 23: OpenAI API configuration

2.7.3 Scripting Engine Implementation

The Script Generation Engine has the task of turning the enriched synthetic user profiles into executable scripts that mimic a real-world interaction of users in a sandbox environment. This engine consists of four different modules that work in concert with each other to generate human-like activity patterns. Each module has distinct function in the pipeline that contributes to difference parts of the script synthesis process.

User profile management module

This module accepts the enriched user profiles as final outputs, that include behavioural metrics and browsing interests. It processes user-specific such as average typing speed, hover time, and scrolling behaviour. These user-specific attributes are passed downstream to affect the realistic simulation of behaviours in the later modules.

```python
import json
from typing import Dict, List, Any, Optional

def load_profile(profile_path: str) -> Dict[str, Any]:
    """Load user profile from JSON file."""
    try:
        with open(profile_path, 'r') as file:
            profile_data = json.load(file)
        print(f"Successfully loaded profile from {profile_path}")
        return profile_data
    except Exception as e:
        print(f"Error loading profile: {e}")
        return {}

def get_mouse_attributes(profile_data: Dict[str, Any]) -> Dict[str, float]:
    """Extract mouse-related attributes from profile."""
    mouse_attrs = {
        'avg_speed': profile_data.get('avg_mouse_speed_px_s', 300),
        'speed_variance': profile_data.get('var_mouse_speed_px_s', 50),
        'avg_distance': profile_data.get('avg_distance_per_movement_px', 200),
        'avg_curvature': profile_data.get('avg_path_curvature', 0.5),
        'curvature_variance': profile_data.get('var_path_curvature', 0.3),
        'hover_time': profile_data.get('avg_hover_time_ms', 1000),
        'hover_variance': profile_data.get('var_hover_time_ms', 200),
        'diagonal_percent': profile_data.get('diagonal_movements_percent', 40),
        'single_click_freq': profile_data.get('single_click_frequency', 50),
        'double_click_freq': profile_data.get('double_click_frequency', 5),
        'right_click_freq': profile_data.get('right_click_frequency', 5),
        'abrupt_stops_freq': profile_data.get('frequency_abrupt_stops_mouse', 15)
    }
    return mouse_attrs

def get_keyboard_attributes(profile_data: Dict[str, Any]) -> Dict[str, float]:
    """Extract keyboard-related attributes from profile."""
    keyboard_attrs = {
        'typing_speed': profile_data.get('typing_speed_wpm', 30),
        'corrections_rate': profile_data.get('corrections_per_100_keys', 10),
        'pause_frequency': profile_data.get('pauses_over_2s', 10),
        'repeated_patterns': profile_data.get('frequency_repeated_text_patterns', 5),
        'spacebar_enter_usage': profile_data.get('frequency_spacebar_enter_usage', 15),
        'caps_lock_usage': profile_data.get('caps_lock_usage', 1),
        'punctuation_usage': profile_data.get('punctuation_usage', 14),
        'shortcuts_usage': profile_data.get('keyboard_shortcuts_usage', 8),
        'avg_word_length': profile_data.get('avg_word_length_chars', 5),
        'word_length_variance': profile_data.get('variability_word_length_chars', 2),
        'common_digraphs_percent': profile_data.get('common_digraphs_percent', 27),
        'common_trigraphs_percent': profile_data.get('common_trigraphs_percent', 13)
    }
    return keyboard_attrs

def get_scroll_attributes(profile_data: Dict[str, Any]) -> Dict[str, float]:
    """Extract scrolling-related attributes from profile."""
    scroll_attrs = {
        'scroll_percent': profile_data.get('scroll_percentage', 70),
        'avg_scroll_length': profile_data.get('avg_scroll_length_px', 250),
        'scroll_events_count': profile_data.get('scroll_events', 20),
        'direction_changes': profile_data.get('scroll_direction_changes', 8)
    }
    return scroll_attrs

def get_browser_attributes(profile_data: Dict[str, Any]) -> Dict[str, Any]:
    """Extract browser behavior attributes from profile."""
    browser_data = profile_data.get('browser_behaviors', {})
    browser_attrs = {
        'search_terms': browser_data.get('search_terms', []),
        'websites': browser_data.get('websites', []),
        'focused_browsing': profile_data.get('focused_browsing_percent', 60)
```

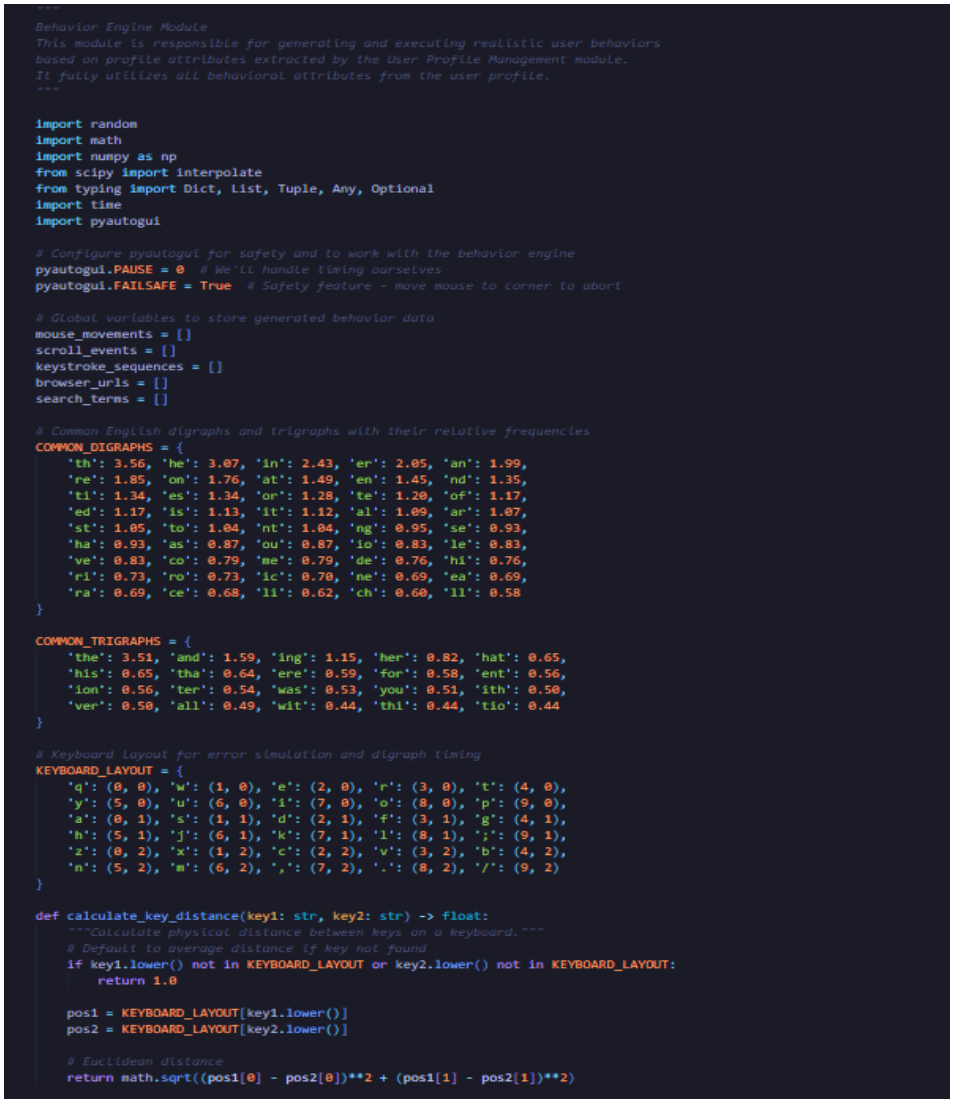
Figure 24: User profile management module implementation

Behaviour Engine

Using the behavioural attributes from the Profile Management Module, the Behaviour Engine generates generalized user action patterns such as:

- Typing sequences with variability and pauses
- Mouse movements including jitter and idle times
- Scroll actions with realistic speed and range

These behaviours are designed to emulate authentic interaction dynamics and are reused across activity templates.

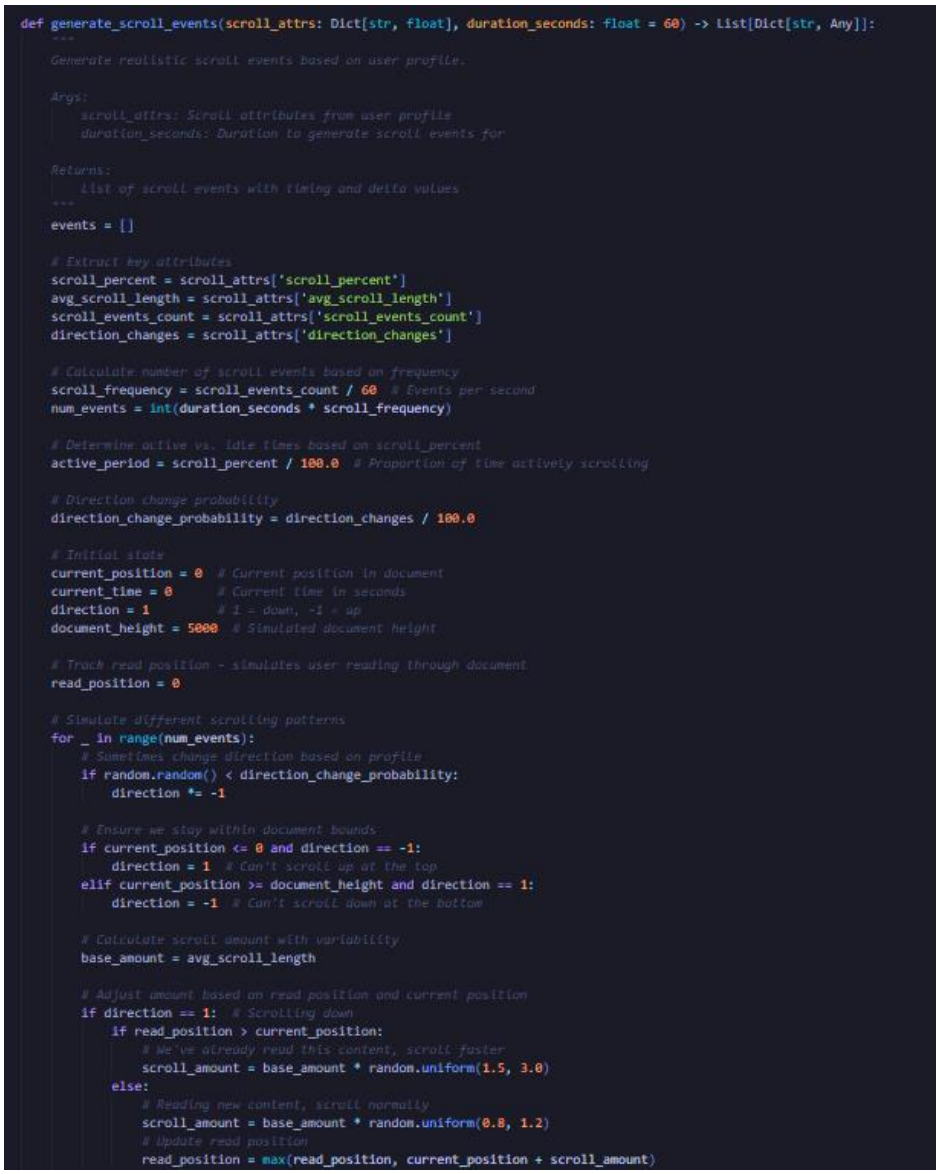


```
"""
Behavior Engine Module
This module is responsible for generating and executing realistic user behaviors
based on profile attributes extracted by the User Profile Management module.
It fully utilizes all behavioral attributes from the user profile.
"""

import random
import math
import numpy as np
from scipy import interpolate
from typing import Dict, List, Tuple, Any, Optional
import time
import pyautogui

# Configure pyautogui for safety and to work with the behavior engine
pyautogui.PAUSE = 0  # We'll handle timing ourselves
pyautogui.FAILSAFE = True  # Safety feature - move mouse to corner to abort

# Global variables to store generated behavior data
mouse_movements = []
scroll_events = []
keystroke_sequences = []
browser_urls = []
search_terms = []

# Common English digraphs and trigraphs with their relative frequencies
COMMON_DIGRAPHS = {
    'th': 3.56, 'he': 3.07, 'in': 2.43, 'er': 2.05, 'an': 1.99,
    're': 1.85, 'on': 1.76, 'at': 1.49, 'en': 1.45, 'nd': 1.35,
    'ti': 1.34, 'es': 1.34, 'or': 1.28, 'te': 1.20, 'of': 1.17,
    'ed': 1.17, 'is': 1.13, 'it': 1.12, 'al': 1.09, 'ar': 1.07,
    'st': 1.05, 'to': 1.04, 'nt': 1.04, 'ng': 0.95, 'se': 0.93,
    'ha': 0.93, 'as': 0.87, 'ou': 0.87, 'io': 0.83, 'le': 0.83,
    've': 0.83, 'co': 0.79, 'me': 0.79, 'de': 0.76, 'hi': 0.76,
    'ri': 0.73, 'ro': 0.73, 'ic': 0.70, 'ne': 0.69, 'ea': 0.69,
    'ra': 0.69, 'ce': 0.68, 'li': 0.62, 'ch': 0.60, 'll': 0.58
}

COMMON_TRIGRAPHS = {
    'the': 3.51, 'and': 1.59, 'ing': 1.15, 'her': 0.82, 'hat': 0.65,
    'his': 0.65, 'tha': 0.64, 'ere': 0.59, 'for': 0.58, 'ent': 0.56,
    'ion': 0.56, 'ter': 0.54, 'was': 0.53, 'you': 0.51, 'ith': 0.50,
    'ver': 0.50, 'all': 0.49, 'wit': 0.44, 'thi': 0.44, 'tio': 0.44
}

# Keyboard layout for error simulation and digraph timing
KEYBOARD_LAYOUT = {
    'q': (0, 0), 'w': (1, 0), 'e': (2, 0), 'r': (3, 0), 't': (4, 0),
    'y': (5, 0), 'u': (6, 0), 'i': (7, 0), 'o': (8, 0), 'p': (9, 0),
    'a': (0, 1), 's': (1, 1), 'd': (2, 1), 'f': (3, 1), 'g': (4, 1),
    'h': (5, 1), 'j': (6, 1), 'k': (7, 1), 'l': (8, 1), ';': (9, 1),
    'z': (0, 2), 'x': (1, 2), 'c': (2, 2), 'v': (3, 2), 'b': (4, 2),
    'n': (5, 2), 'm': (6, 2), ',': (7, 2), '.': (8, 2), '/': (9, 2)
}

def calculate_key_distance(key1: str, key2: str) -> float:
    """Calculate physical distance between keys on a keyboard."""
    # Default to average distance if key not found
    if key1.lower() not in KEYBOARD_LAYOUT or key2.lower() not in KEYBOARD_LAYOUT:
        return 1.0

    pos1 = KEYBOARD_LAYOUT[key1.lower()]
    pos2 = KEYBOARD_LAYOUT[key2.lower()]

    # Euclidean distance
    return math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)
```

Figure 25: Behaviour engine module implementation

```python
def generate_mouse_movements(mouse_attrs: Dict[str, float], count: int = 15) -> List[Dict[str, Any]]:
    """
    Generate realistic mouse movement paths based on profile attributes.

    Args:
        mouse_attrs: Mouse attributes from user profile
        count: Number of movement patterns to generate

    Returns:
        List of movement patterns with paths and timings
    """
    movements = []

    # Extract key attributes
    avg_speed = mouse_attrs['avg_speed']
    speed_variance = mouse_attrs['speed_variance']
    avg_curvature = mouse_attrs['avg_curvature']
    curvature_variance = mouse_attrs['curvature_variance']
    diagonal_percent = mouse_attrs['diagonal_percent']
    abrupt_stops_freq = mouse_attrs['abrupt_stops_freq']
    hover_time = mouse_attrs['hover_time'] / 1000.0  # Convert ms to seconds
    hover_variance = mouse_attrs['hover_variance'] / 1000.0  # Convert ms to seconds

    # Generate various movement patterns
    for i in range(count):
        # Decide if this is a diagonal-favoring movement based on profile
        is_diagonal = random.random() < (diagonal_percent / 100.0)

        # Create start and end points with directional bias if diagonal
        screen_width, screen_height = pyautogui.size()
        start_x = random.randint(0, screen_width)
        start_y = random.randint(0, screen_height)

        if is_diagonal:
            # Create more diagonal movement
            angle = random.uniform(0, 2 * math.pi)
            distance = random.uniform(100, 500)
            # Bias angle towards 45° increments for diagonals
            angle = round(angle / (math.pi/4)) * (math.pi/4)
            end_x = min(max(0, start_x + distance * math.cos(angle)), screen_width)
            end_y = min(max(0, start_y + distance * math.sin(angle)), screen_height)
        else:
            # More random movement
            end_x = random.randint(0, screen_width)
            end_y = random.randint(0, screen_height)

        # Calculate distance
        distance = math.sqrt((end_x - start_x)**2 + (end_y - start_y)**2)

        # Determine number of control points based on distance and curvature
        points_count = max(3, int(distance / 100))

        # Apply curvature based on profile
        curvature = random.gauss(avg_curvature, curvature_variance)
        curvature = max(0.1, min(0.9, curvature))  # Constrain to reasonable values

        # Create control points for curve
        control_points_x = [start_x]
        control_points_y = [start_y]

        # Generate middle control points with randomization
        for j in range(1, points_count - 1):
            # Linear interpolation
            t = j / (points_count - 1)
            x = start_x + t * (end_x - start_x)
            y = start_y + t * (end_y - start_y)
```

Figure 26: Behaviour engine module implementation



```python
def generate_scroll_events(scroll_attrs: Dict[str, float], duration_seconds: float = 60) -> List[Dict[str, Any]]:
    """
    Generate realistic scroll events based on user profile.

    Args:
        scroll_attrs: Scroll attributes from user profile
        duration_seconds: Duration to generate scroll events for

    Returns:
        List of scroll events with timing and delta values
    """
    events = []

    # Extract key attributes
    scroll_percent = scroll_attrs['scroll_percent']
    avg_scroll_length = scroll_attrs['avg_scroll_length']
    scroll_events_count = scroll_attrs['scroll_events_count']
    direction_changes = scroll_attrs['direction_changes']

    # Calculate number of scroll events based on frequency
    scroll_frequency = scroll_events_count / 60  # Events per second
    num_events = int(duration_seconds * scroll_frequency)

    # Determine active vs. idle times based on scroll_percent
    active_period = scroll_percent / 100.0  # Proportion of time actively scrolling

    # Direction change probability
    direction_change_probability = direction_changes / 100.0

    # Initial state
    current_position = 0  # Current position in document
    current_time = 0      # Current time in seconds
    direction = 1         # 1 = down, -1 = up
    document_height = 5000  # Simulated document height

    # Track read position - simulates user reading through document
    read_position = 0

    # Simulate different scrolling patterns
    for _ in range(num_events):
        # Sometimes change direction based on profile
        if random.random() < direction_change_probability:
            direction *= -1

        # Ensure we stay within document bounds
        if current_position <= 0 and direction == -1:
            direction = 1  # Can't scroll up at the top
        elif current_position >= document_height and direction == 1:
            direction = -1  # Can't scroll down at the bottom

        # Calculate scroll amount with variability
        base_amount = avg_scroll_length

        # Adjust amount based on read position and current position
        if direction == 1:  # Scrolling down
            if read_position > current_position:
                # We've already read this content, scroll faster
                scroll_amount = base_amount * random.uniform(1.5, 3.0)
            else:
                # Reading new content, scroll normally
                scroll_amount = base_amount * random.uniform(0.8, 1.2)
                # Update read position
                read_position = max(read_position, current_position + scroll_amount)
```

Figure 27: Behaviour engine module implementation

Scheduler Module

The scheduler is responsible for temporal coordination. It takes the full set of enriched activity scripts and assigns them to appropriate time slots based on a generated timetable. This ensures that script execution mimics a plausible user workday or activity schedule.

```python
"""
Scheduler & Execution Module
This module is responsible for creating and managing execution schedules for behavior scripts.
"""

import os
import json
import random
import shutil
from typing import Dict, List, Any, Optional
import time

def create_schedule(template_names: List[str], duration_minutes: int = 60) -> Dict[str, Any]:
    """
    Create a schedule for script execution over specified duration.

    Args:
        template_names: List of available template names
        duration_minutes: Total duration in minutes

    Returns:
        A schedule dictionary with activities and their timings
    """
    if not template_names:
        raise ValueError("No templates available for scheduling")

    # Determine how many activities to include based on duration and available templates
    min_activities = max(1, duration_minutes // 15)  # At least 1 activity per 15 minutes
    max_activities = max(min_activities * 2, duration_minutes // 5)  # At most 1 activity per 5 minutes

    # Determine number of activities
    num_activities = min(max_activities, random.randint(min_activities, max_activities))

    # If we have fewer templates than needed activities, we'll reuse templates
    needed_templates = []
    while len(needed_templates) < num_activities:
        # Add templates in random order
        shuffled = random.sample(template_names, min(num_activities - len(needed_templates), len(template_names)))
        needed_templates.extend(shuffled)

    # Start and end times
    start_time = 0  # in seconds
    end_time = duration_minutes * 60  # in seconds

    # Create schedule
    schedule = {
        "duration_minutes": duration_minutes,
        "activities": []
    }

    # Current time pointer
    current_time = start_time

    # Distribute activities across the schedule
    for template_name in needed_templates:
        # Determine activity duration based on remaining time
        remaining_time = end_time - current_time
        if remaining_time <= 0:
            break  # No more time left

        # Calculate max duration to ensure we don't exceed the schedule
        max_duration = min(10 * 60, remaining_time)  # Max 10 minutes or remaining time
        min_duration = min(1 * 60, max_duration)     # Min 1 minute unless less time remains

        # Determine activity duration
        activity_duration = random.randint(min_duration, max_duration)
```

Figure 28: Scheduler module implementation

```python
def export_schedule(schedule: Dict[str, Any], output_dir: str,
                    filename: str = "schedule.json") -> str:
    """
    Export the schedule to a JSON file.

    Args:
        schedule: The schedule to export
        output_dir: Directory to save the file
        filename: Name of the JSON file

    Returns:
        Path to the exported schedule file
    """
    os.makedirs(output_dir, exist_ok=True)
    output_path = os.path.join(output_dir, filename)

    try:
        with open(output_path, 'w') as file:
            json.dump(schedule, file, indent=4)
        print(f"Schedule exported to {output_path}")
        return output_path
    except Exception as e:
        print(f"Error exporting schedule: {e}")
        return ""

def prepare_scripts(schedule: Dict[str, Any], template_contents: Dict[str, str],
                    output_dir: str, mouse_movements: List[Dict[str, Any]],
                    scroll_events: List[Dict[str, Any]],
                    keystroke_sequences: List[List[Dict[str, Any]]],
                    browser_urls: List[str], search_terms: List[str]) -> List[str]:
    """
    Prepare all scripts based on the schedule.

    Args:
        schedule: The execution schedule
        template_contents: Dictionary of template names to content
        output_dir: Directory to save generated scripts
        mouse_movements, scroll_events, keystroke_sequences: Behavior data
        browser_urls, search_terms: Browser data

    Returns:
        List of paths to prepared script files
    """
    from script_manager import generate_script

    prepared_scripts = []

    # Create output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    # Count number of templates actually available
    available_templates = set(template_contents.keys())
    required_templates = set(activity["template"] for activity in schedule["activities"])
    missing_templates = required_templates - available_templates

    if missing_templates:
        print(f"Warning: {len(missing_templates)} template(s) required by the schedule are not available:")
        for template in missing_templates:
            print(f"  - {template}")

    # Generate and save each script in the schedule
    for activity in schedule["activities"]:
        template_name = activity["template"]
```

Figure 29: Scheduler module implementation

2.7.4 WGAN-GP Training Process Testing

The first stage of testing involved verifying that the GAN model was implemented and trained correctly without interruption. The model was trained using the real,

preprocessed behavioural dataset. During training, both the generator and critic were updated using their respective loss functions, and early stopping was implemented to preserve the best-performing model.

```
Epoch 50/2000 | G Loss: 1.2428 | C Loss: 0.2385 | Avg KS p-value: 0.078973 | Time: 32.4s
New best model saved with avg p-value: 0.078973
Epoch 100/2000 | G Loss: 3.2555 | C Loss: -0.1414 | Avg KS p-value: 0.038452 | Time: 60.6s
Epoch 150/2000 | G Loss: 4.3731 | C Loss: -0.0927 | Avg KS p-value: 0.003238 | Time: 87.8s
Epoch 200/2000 | G Loss: 5.5302 | C Loss: 0.0762 | Avg KS p-value: 0.021210 | Time: 114.7s
Epoch 250/2000 | G Loss: 4.7568 | C Loss: -0.1199 | Avg KS p-value: 0.011818 | Time: 141.5s
Epoch 300/2000 | G Loss: 4.8916 | C Loss: 0.0195 | Avg KS p-value: 0.001704 | Time: 168.5s
```

Figure 30: WGAN-GP training process

After the training process concluded (either after full epochs or due to early stopping), the best generator model was restored from its checkpoint. Using this model, a batch of synthetic profiles was generated by feeding latent vectors through the trained generator.


```
🔍 Generated User Profile:
[
    {
        "avg_mouse_speed_px_s":373.4039066505,
        "var_mouse_speed_px_s":55.240733552,
        "total_mouse_distance_px":3592.4276051402,
        "avg_distance_per_movement_px":224.0663007203,
        "avg_path_curvature":0.4781393528,
        "var_path_curvature":0.332795608,
        "avg_hover_time_ms":1086.7153979743,
        "var_hover_time_ms":233.8082518566,
        "avg_scroll_length_px":251.2282219198,
        "diagonal_movements_percent":40.8620772552,
        "avg_word_length_chars":6.7604810506,
        "variability_word_length_chars":2.2121785879,
        "short_words_percent":37.620656569,
        "medium_words_percent":55.5363440514,
        "long_words_percent":6.1098894477,
        "capitalized_words_percent":9.178208828,
        "scroll_percentage":70.4458377844,
        "focused_browsing_percent":62.9373497233,
        "single_click_frequency":49.3219870329,
        "double_click_frequency":5.8581960201,
        "right_click_frequency":5.8352919817,
        "scroll_events":21.3023298979,
        "frequency_abrupt_stops_mouse":14.4844514132,
        "typing_speed_wpm":29.0968880057,
        "corrections_per_100_keys":11.2836056948,
        "pauses_over_2s":13.5629764199,
        "frequency_repeated_text_patterns":6.121175766,
        "frequency_spacebar_enter_usage":14.4753834605,
        "caps_lock_usage":1.307477206,
        "punctuation_usage":14.0154078603,
        "total_digraphs":139.6879136562,
        "common_digraphs_percent":27.2695434093,
        "total_trigraphs":173.7811312079,
        "common_trigraphs_percent":13.3035737276,
        "words_typed_per_session":475.0028371811,
        "mouse_stopping_events":12.8840005398,
        "scroll_direction_changes":8.2555548847,
        "keyboard_shortcuts_usage":8.5014643669
    }
]
```


Figure 31: Generated user profile by WGAN-GP model

2.7.5 OpenAI Enrichment Testing

After the GAN model generates a synthetic user profile, the next step is to enrich the profile with interest-based web activity.

It is expected for the API's response to contain two items for each interest:-

- A list of search terms that a user might enter
- A list of realistic URLs or websites that relate to the topic

Figure 32: OpenAI response

2.7.6 Scripting Engine Testing

The scripting engine was launched using a generated synthetic user profile as input. Upon execution, the system sequentially activated each of its core modules—parsing the profile, generating generalised behaviour patterns (mouse movements, keystrokes, scrolling), updating script templates, and creating the session schedule.

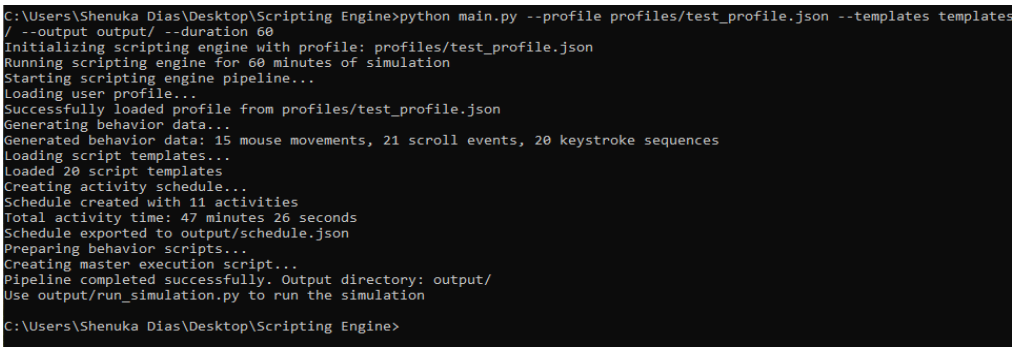
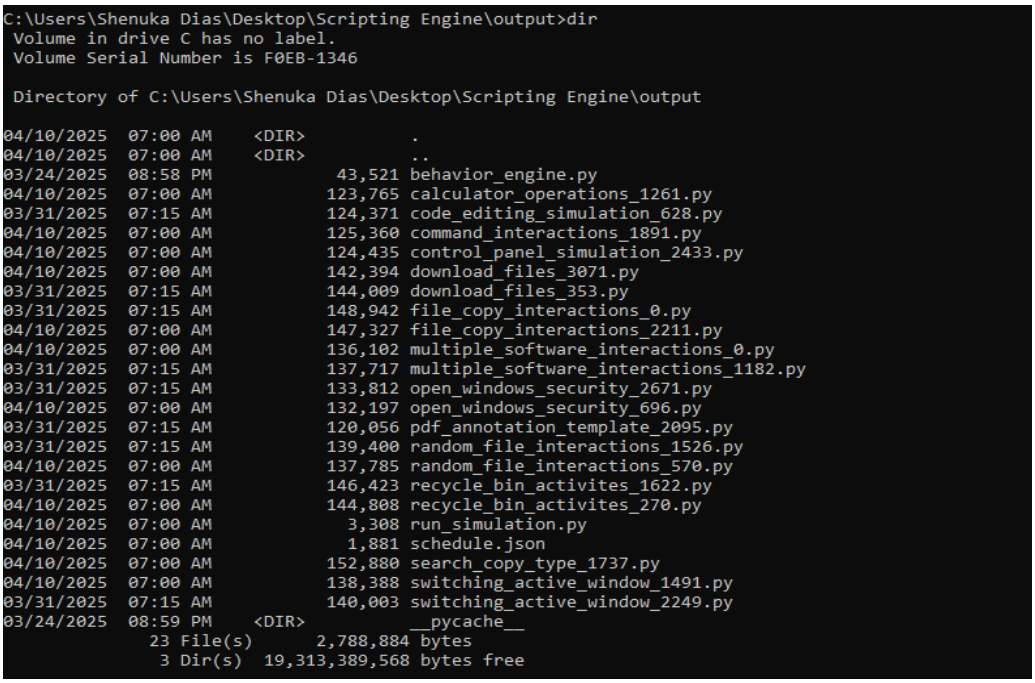


Figure 33: Scripting engine execution

The engine correctly populated user behaviour templates using data from the Behaviour Engine. Each selected template was updated with timing and movement logic based on the user's typing speed, scroll rate, and mouse activity.

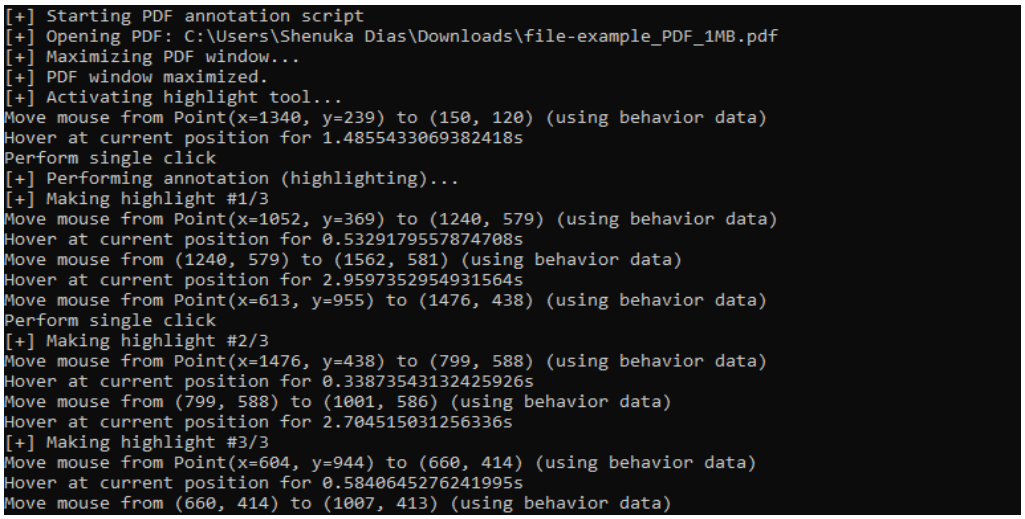
Below figure 34 shows the scripts that were generated for the user session.



```
C:\Users\Shenuka Dias\Desktop\Scripting Engine\output>dir
 Volume in drive C has no label.
 Volume Serial Number is F0EB-1346

 Directory of C:\Users\Shenuka Dias\Desktop\Scripting Engine\output

04/10/2025  07:00 AM    <DIR>          .
04/10/2025  07:00 AM    <DIR>          ..
03/24/2025  08:58 PM            43,521 behavior_engine.py
04/10/2025  07:00 AM           123,765 calculator_operations_1261.py
03/31/2025  07:15 AM           124,371 code_editing_simulation_628.py
04/10/2025  07:00 AM           125,360 command_interactions_1891.py
04/10/2025  07:00 AM           124,435 control_panel_simulation_2433.py
04/10/2025  07:00 AM           142,394 download_files_3071.py
03/31/2025  07:15 AM           144,009 download_files_353.py
03/31/2025  07:15 AM           148,942 file_copy_interactions_0.py
04/10/2025  07:00 AM           147,327 file_copy_interactions_2211.py
04/10/2025  07:00 AM           136,102 multiple_software_interactions_0.py
03/31/2025  07:15 AM           137,717 multiple_software_interactions_1182.py
03/31/2025  07:15 AM           133,812 open_windows_security_2671.py
04/10/2025  07:00 AM           132,197 open_windows_security_696.py
03/31/2025  07:15 AM           120,056 pdf_annotation_template_2095.py
03/31/2025  07:15 AM           139,400 random_file_interactions_1526.py
04/10/2025  07:00 AM           137,785 random_file_interactions_570.py
03/31/2025  07:15 AM           146,423 recycle_bin_activites_1622.py
04/10/2025  07:00 AM           144,808 recycle_bin_activites_270.py
04/10/2025  07:00 AM             3,308 run_simulation.py
04/10/2025  07:00 AM             1,881 schedule.json
04/10/2025  07:00 AM           152,880 search_copy_type_1737.py
04/10/2025  07:00 AM           138,388 switching_active_window_1491.py
03/31/2025  07:15 AM           140,003 switching_active_window_2249.py
03/24/2025  08:59 PM    <DIR>          __pycache__
              23 File(s)      2,788,884 bytes
               3 Dir(s)  19,313,389,568 bytes free
```

Figure 34: Generated user behaviour simulation scripts

Next the scripts were tested execution within the sandbox. Figure 35 shows the logs that got while executing the scripts in the sandbox.



```
[+] Starting PDF annotation script
[+] Opening PDF: C:\Users\Shenuka Dias\Downloads\file-example_PDF_1MB.pdf
[+] Maximizing PDF window...
[+] PDF window maximized.
[+] Activating highlight tool...
Move mouse from Point(x=1340, y=239) to (150, 120) (using behavior data)
Hover at current position for 1.4855433069382418s
Perform single click
[+] Performing annotation (highlighting)...
[+] Making highlight #1/3
Move mouse from Point(x=1052, y=369) to (1240, 579) (using behavior data)
Hover at current position for 0.53291795578747085
Move mouse from (1240, 579) to (1562, 581) (using behavior data)
Hover at current position for 2.95973529549315645
Move mouse from Point(x=613, y=955) to (1476, 438) (using behavior data)
Perform single click
[+] Making highlight #2/3
Move mouse from Point(x=1476, y=438) to (799, 588) (using behavior data)
Hover at current position for 0.33873543132425926s
Move mouse from (799, 588) to (1001, 586) (using behavior data)
Hover at current position for 2.7045150312563365
[+] Making highlight #3/3
Move mouse from Point(x=604, y=944) to (660, 414) (using behavior data)
Hover at current position for 0.5840645276241995s
Move mouse from (660, 414) to (1007, 413) (using behavior data)
```

Figure 35: Successful script execution in sandbox

2.8 Commercialization Aspects

2.8.1 Real World Use Cases

This system responds to the need for more sophisticated user behaviour simulation capabilities in dynamic malware analysis spaces. In traditional sandboxes, the lack of realistic user activity allows more advanced malware to detect it is in a sandbox and escape analysis. This project will introduce a platform that creates synthetic user profiles and automates human-like interaction patterns to increase the realism of sandbox environments.

Real-world use cases include:

- Enterprise malware analysis platforms that require realistic interaction to uncover evasive threats.

- Threat intelligence teams seeking deeper insights by simulating diverse user scenarios.

- Security research labs conducting behavioural malware studies in controlled but lifelike settings.

- Government cyber defence units performing adversary emulation and APT testing.

2.8.2 Market Need and Relevance

The potential for this system commerciality is based in part on the increase of sandbox-resistant malware being utilized by cybercriminals and advanced persistent threat (APT) groups. Many of the existing sandboxing solutions do not have fidelity of user interaction and therefore cannot be used to their full potential. As malware increases in its context awareness, the way to accomplish this is by providing tools that can

convincingly simulate the behavior of a real user in order to prompt the execution of the malware in entirety.

3. RESULTS & DISCUSSION

3.1. Results

3.1.1 GAN Training Metrics

The WGAN-GP model was trained over a total of 2000 epochs, with periodic evaluation checkpoints. The training process focused on balancing the adversarial interaction between the Generator and Critic networks, while maintaining statistical similarity with real-world behavioural profiles.

Loss Curves

As shown in figure 36, the Generator loss exhibited a rising trend as it learned to produce more realistic samples. The Critic loss remained relatively stable after initial fluctuations, indicating consistent gradient feedback. These patterns are typical of successful WGAN-GP training.

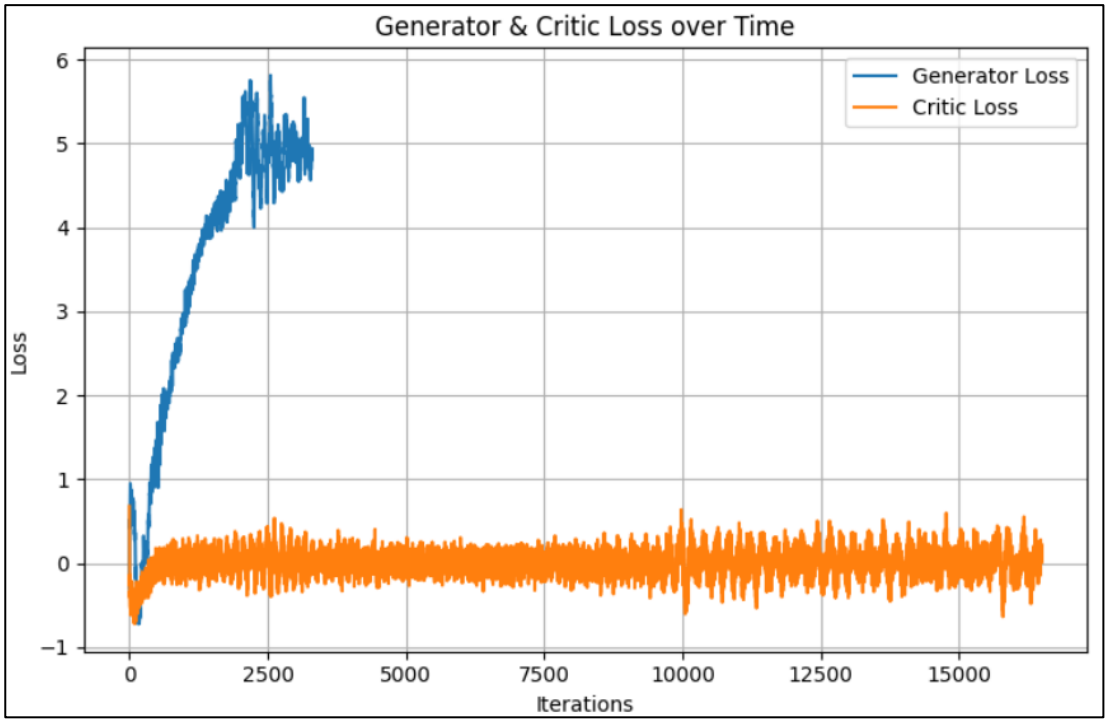


Figure 36: Generator & critic loss over time

KS Test Evaluation

Figure 37 illustrates the average Kolmogorov–Smirnov (KS) test p-value over training. A downward trend was observed, signifying improved alignment between synthetic and real feature distributions. The best p-value achieved was approximately 0.0082 at epoch 300, after which early stopping was triggered.

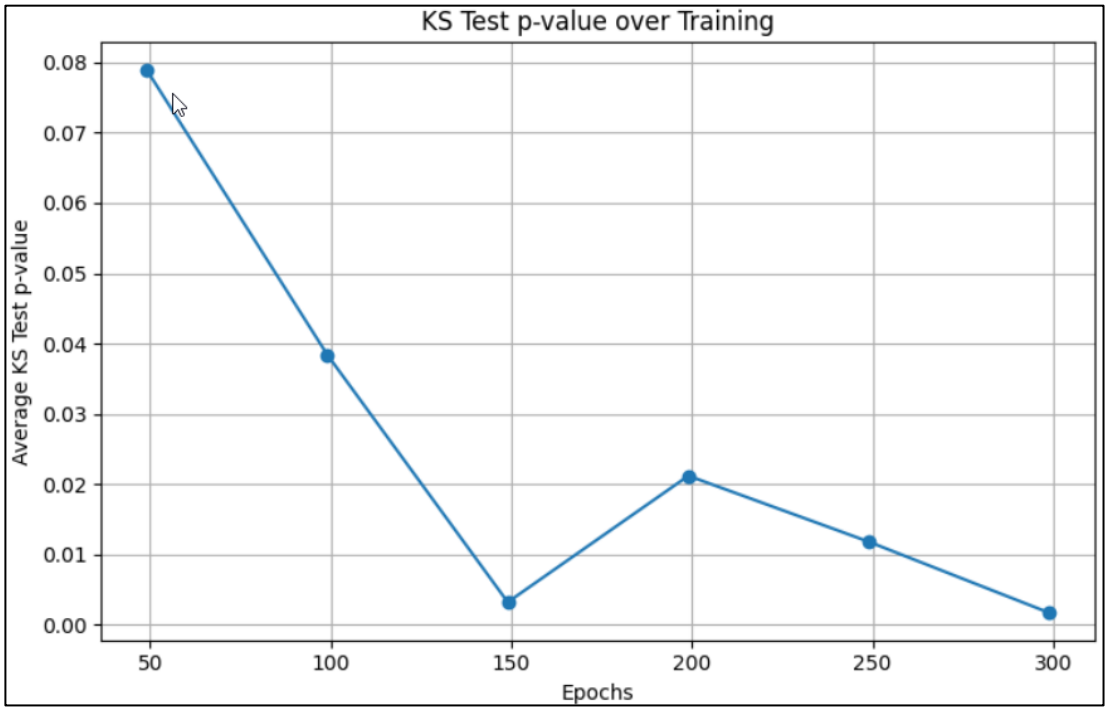


Figure 37: KS test values over training

3.1.2 Feature-wise Distribution Alignment (KS Test)

The Kolmogorov-Smirnov (KS) test was used to examine the statistical similarity between the real and generated user behaviour data for each feature. This non-parametric test compares the cumulative distributions of two samples to assess whether both samples came from the same distribution.

The KS p-values that were obtain reveal how closely synthetic data mimic the original behaviour data at the feature level. Generally, a higher p-value corresponds to a closer resemblance in distributional shape of the original behaviour and generated behaviour, whereas a lower p-value corresponds to greater divergence.

This analysis has great value as a (quantitative) measure of our generative modelling to produce realistic user behaviour behaviours across a range of different behaviours, and thus provides a guide to the strengths of behavioural representations and areas for refinement.

index	KS Test P-Value
caps_lock_usage	5.592162145443055e-19
mouse_stopping_events	3.100491786591616e-11
frequency_repeated_text_patterns	3.310424729932166e-10
corrections_per_100_keys	5.2030715224636835e-09
keyboard_shortcuts_usage	2.6393297395044085e-08
frequency_spacebar_enter_usage	5.505136382354229e-07
common_trigraphs_percent	9.280869260288718e-07
frequency_abrupt_stops_mouse	1.1023110210119265e-06
common_digraphs_percent	8.671537885149037e-05
total_digraphs	8.987089889683542e-05
right_click_frequency	0.0001812379542543572 5
var_hover_time_ms	0.0002302752039352981 7
double_click_frequency	0.0003682689824168959
avg_path_curvature	0.001093368957843272
punctuation_usage	0.0013554686172590122
pauses_over_2s	0.002063773121733815
var_path_curvature	0.0021256662238033903
scroll_direction_changes	0.0029297551710701884
total_mouse_distance_px	0.014355265370609601
total_trigraphs	0.01930185590746952
avg_word_length_chars	0.022843850936268898
typing_speed_wpm	0.025117270550781348
avg_mouse_speed_px_s	0.03975125933462197
short_words_percent	0.044408594293862305
single_click_frequency	0.053992498471931556
focused_browsing_percent	0.061329007639531447
avg_scroll_length_px	0.06807931121076935
scroll_events	0.07392580199690227
scroll_percentage	0.09784438737213373
variability_word_length_chars	0.09784438737213373
diagonal_movements_percent	0.12318255239107306
medium_words_percent	0.12788739791848683

Figure 38: KS test values

While the individual feature scores vary, the overall KS test results demonstrate that most of the behavioural attributes in the synthetic profiles generally align closely with real values of behaviour penetration. This is further indicative of the WGAN-GP with correlation loss and post processing modifications being used to capture dense and complex behavioural patterns effectively. While several of the attributes did register lower p-values thus suggesting distributional gaps, this was also expected based on

their variable and sparse nature. Regardless, the synthetic generated profiles exhibit statistical credibility to actual user behaviour, thus advocating confidence in the usability of the synthetic data for downstream uses.

3.1.3 Feature-Wise Visual Comparison

To provide additional assurance of the quality of the produced synthetic data, the results were visually compared the distribution of selected key behavioural attributes in the observed data and the generated data. Histograms were plotted for features such as the average mouse speed, typing speed, and scroll frequency. Although these visualizations can be primarily viewed as common sense replication of real-world behaviours, comparing the histograms allowed to visualize data qualities at the same time. Overlap between the real and synthetic curves confirms that while a part of the real-world distribution can be very different from actual observed behaviours, the overall model generated behaviours that preserve underlying properties of the distributions.

Figure 39: Visual comparison average mouse speed

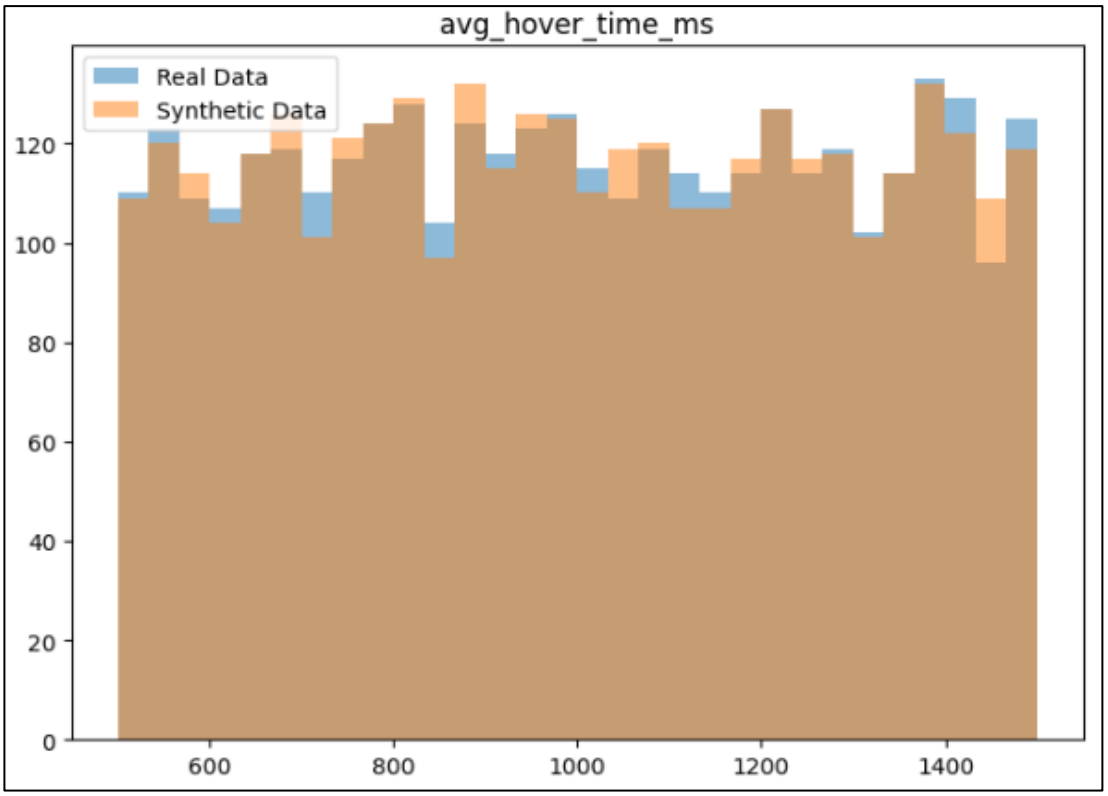


Figure 40: Visual comparison average hover time

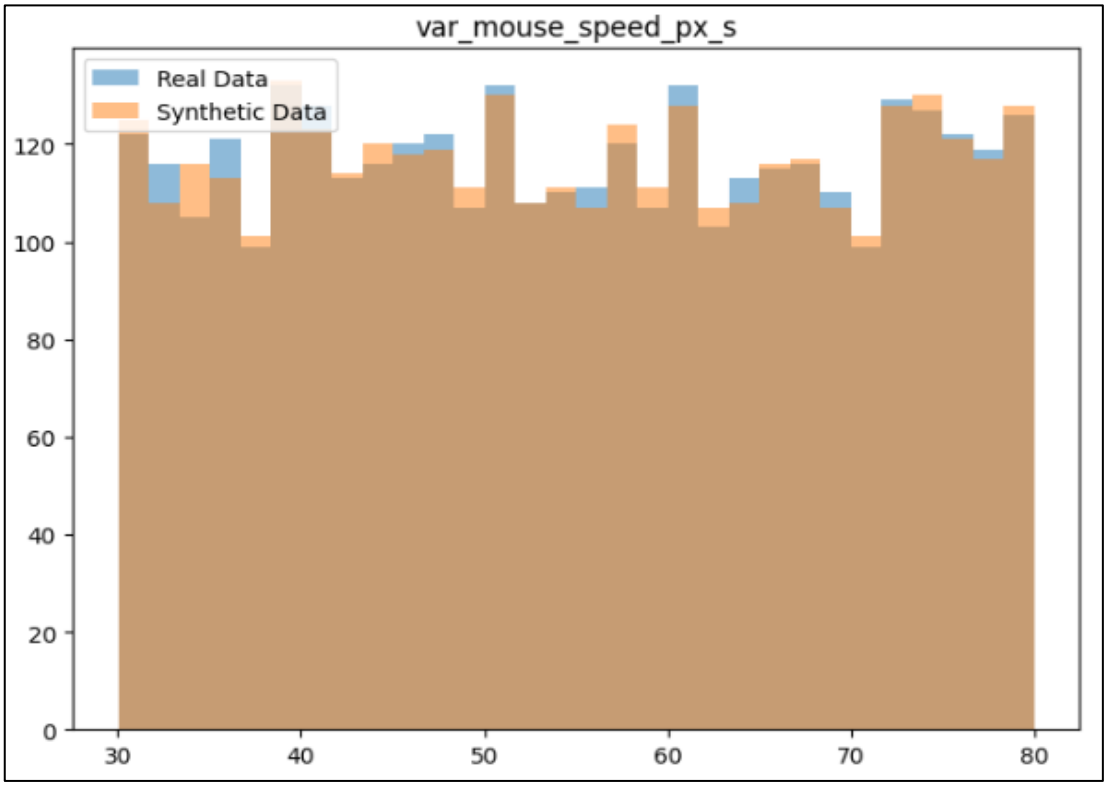


Figure 41: Visual comparison variance of mouse speed

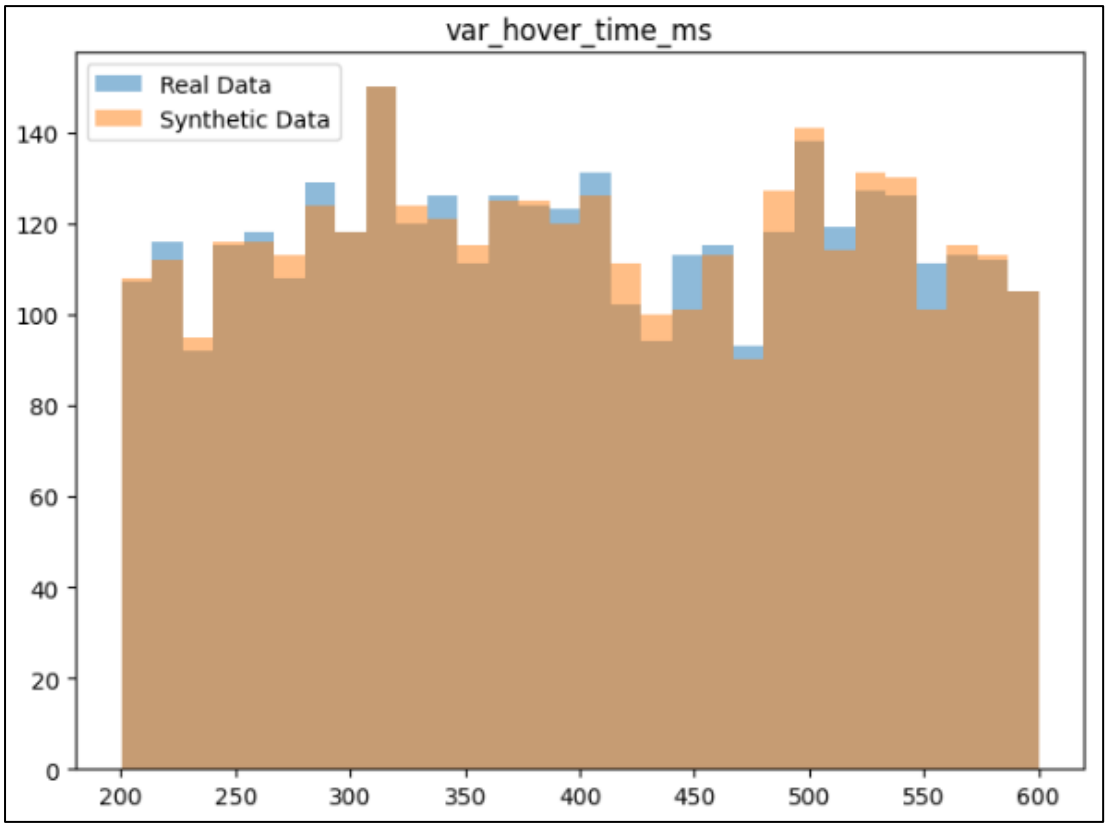


Figure 43: Visual comparison variance of hover time

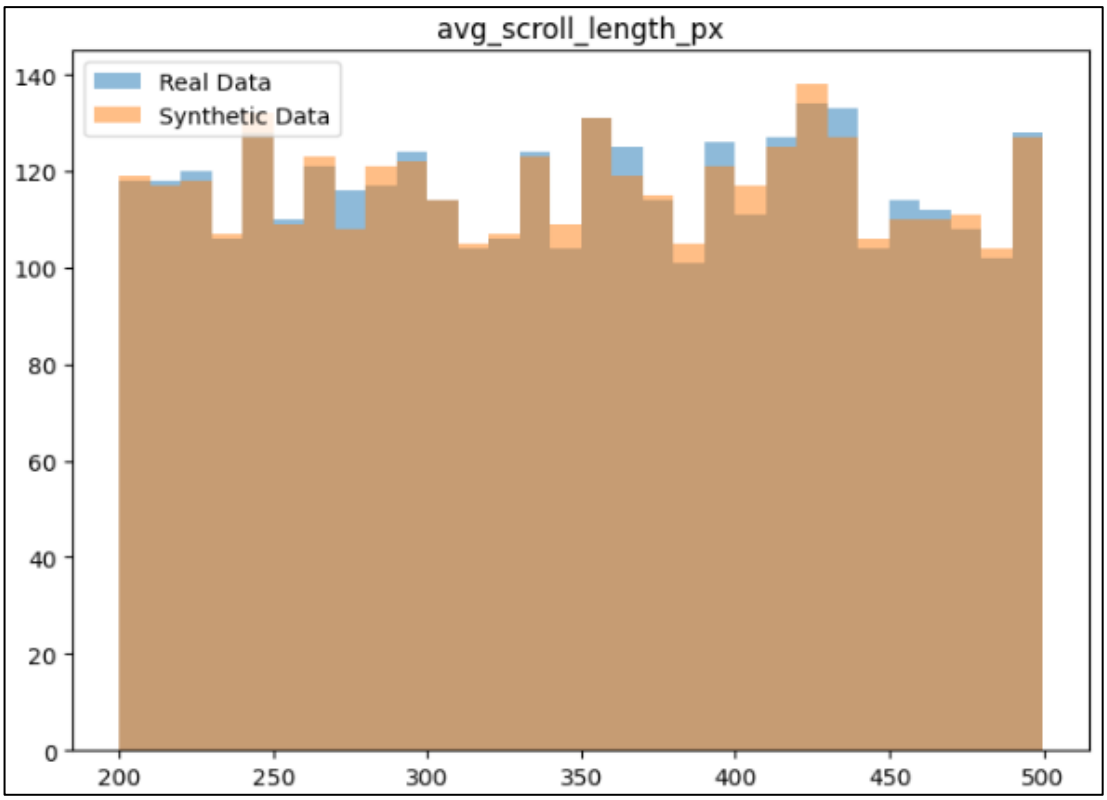


Figure 42: Visual comparison of average scroll length

3.1.4 Cuckoo Sandbox Results

To evaluate the effectiveness of the proposed user behaviour simulation in uncovering sandbox-evasive malware, a set of malware samples were executed in a controlled sandbox environment. These malware samples were not randomly chosen but were generated by the offensive component of this research project. That component leverages Reinforcement Learning (RL) to produce sandbox-aware malware specifically designed to detect artificial execution environments and suppress malicious behaviour unless realistic user interactions are observed.

Figure 44: Sandbox results before user behaviour simulation

Figure 45: Sandbox results after user behaviour simulation

The figures above presents the execution scores under two scenarios. The figure 44 shows results from malware executed without simulated user activity. As expected, the malware largely refrains from exhibiting malicious behaviour, leading to low detection scores ranging from 0.0 to 2.4. This confirms their evasive nature and ability to identify a sandboxed context.

In contrast, the figure 45 shows scores after applying the proposed user behaviour simulation within the sandbox. Here, the malware is successfully triggered, leading to significantly higher scores (up to 5.2). This highlights the capability of the user simulation framework to bypass sandbox detection heuristics, forcing evasive malware to reveal its true behavioural traits during analysis.

3.2. Research Findings

The results from the conducted experiments highlight the significant progress made in generating and simulating realistic user behaviour within sandbox environments.

3.2.1 Effectiveness of Synthetic Profile Generation

The WGAN-GP architecture has proven very efficient at learning and reproducing the statistical properties of actual user behaviour data. The inclusion of correlation loss guarantees that cross-feature relationships in the synthetic data closely resembled those identified in the real data. Furthermore, postprocessing strategies, specifically denormalisation and distribution matching, effectively adjusted any scale anomalies, and the resultant synthetic profiles produced values that fit within a valid range. The KS test results for a variety of behaviour metrics established the statistical close association between real and generated data, providing good assurance that the model is reliable at synthesising human-like behavioural profiles.

3.2.2 Impact of Behaviour Simulation in Sandbox Environments

A notable difference was observed in malware execution patterns when simulated user activity was introduced into the sandbox. In test runs where no user behaviour was present, many of the sandbox-evasive malware samples exhibited low or zero behavioural scores, indicating successful evasion. However, when realistic user behaviours were emulated through the scripted profiles, the same samples triggered significantly higher detection scores. This shift clearly demonstrates that malware can detect the absence of human interaction and remain dormant thus reinforcing the need for realistic user simulation in dynamic analysis environments.

4. CONCLUSION

Particularly with the emergence of sandbox-evasive techniques, the increasing complexity of malware has shown important limits in conventional sandbox-based analysis. Many contemporary malware samples remain dormant or show benign behavior when executed in sandbox environments where user behaviour is not available. This evasiveness causes missed detections in practical situations and compromises the efficacy of dynamic analysis techniques. Dealing with this problem, the shown in this article offers a new approach to fool and activate such evasive malware by including generative user simulation into the sandboxing process.

A key contribution of this work is the use of a Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) to generate realistic behavioural profiles. These profiles are statistically aligned with real user data and include key interaction features such as mouse movements, keystroke timing, digraph and trigraph dynamics. The generated profiles are further enriched with interest-specific context using OpenAI's large language models, which provide relevant search terms and URLs, adding an additional layer of behavioural realism.

The final output a complete, enriched user profile is processed by a modular scripting engine, which automates user interaction patterns through a series of behaviour templates. These templates are dynamically updated based on the user's behavioural attributes and scheduled via a timetable engine. The complete set of scripts, accompanied by the execution schedule, is deployed to the sandbox environment to simulate a fully active user session.

Based on testing, the system displayed a clear functional advantage. In scenarios where there was no simulated user behaviour, the evasive malware samples were inactive and had low detection scores. However, once the synthetic user simulation was installed, the same samples displayed full malicious behaviour and dramatically improved

detection scores. This supports the system's main hypothesis--realistic user simulation can bypass the type of sandbox detection heuristics used by evasive malware.

In addition to its effectiveness, the system is scalable, modular, and automation-friendly, making it suitable for integration into existing malware analysis pipelines or commercial sandbox products. It provides a viable defence strategy that is not reliant on signature-based detection, heuristics, or predefined rules. Instead, it shifts the paradigm towards deception—convincing malware that it is running in a real environment.

This research not only contributes a working prototype but also opens new avenues in cybersecurity defence, where AI-driven user simulation can play a central role in exposing threats that would otherwise remain hidden. The integration of generative models into behavioural simulation represents a significant advancement in the way sandboxes are designed and utilized, positioning this system as a promising solution in the ongoing arms race against advanced malware threats.

5. REFERENCE

[1] A. Mohanta and A. Saldanha, *Malware analysis and detection engineering: A comprehensive approach to detect and analyze modern malware*, 1st ed. Berlin, Germany: APress, 2020.

[2] M. Sikorski and A. Honig, *Practical malware analysis: The hands-on guide to dissecting malicious software*. No Starch Press, 2012.

[3] A. Chailytko and S. Skuratovich, "Defeating Sandbox Evasion How to increase the successful emulation rate in virtual environment," 2016.

[4] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting Environment-Sensitive Malware," in *Lecture Notes in Computer Science*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 338–357.

[5] A. Mills and P. Legg, "Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques," *J. Cybersecur. Priv.*, vol. 1, no. 1, pp. 19–39, 2020.

[6] A. Lindorin, *Impersonating a sandbox against evasive malware*. 2022.

[7] C. Xie *et al.*, "EnvFaker: A method to reinforce Linux sandbox based on tracer, filter and emulator against environmental-sensitive malware," in *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2021.

[8] P. Feng, J. Sun, S. Liu, and K. Sun, "UBER: Combating sandbox evasion via user behaviour emulators," in *Information and Communications Security*, Cham: Springer International Publishing, 2020, pp. 34–50.

[9] S. Liu, P. Feng, S. Wang, K. Sun, and J. Cao, "Enhancing malware analysis sandboxes with emulated user behaviour," *Comput. Secur.*, vol. 115, no. 102613, p. 102613, 2022.

[10] L. Wang *et al.*, "User behaviour simulation with large language model based agents," *arXiv [cs.IR]*, 2023.

[11] Y. Lu and J. Li, "Generative adversarial network for improving deep learning based malware classification," in *2019 Winter Simulation Conference (WSC)*, 2019.

[12] A. Dunmore, J. Jang-Jaccard, F. Sabrina, and J. Kwak, “Generative Adversarial Networks for malware detection: A survey,” *arXiv [cs.CR]*, 2023.