A SYNTHETIC USER ENVIRONMENT FOR NETWORK COUNTER-SURVEILLANCE OPERATIONS

Automatic Generation of Human Interface Device Events

UN ENVIRONNEMENT UTILISATEUR SYNTHÉTIQUE POUR OPÉRATIONS DE CONTRE-SURVEILLANCE RÉSEAU

Génération automatique d’évènements de dispositifs d’interface humaine

A Thesis Submitted to the Division of Graduate Studies
of the Royal Military College of Canada
by

Sylvain Paul Leblanc, CD, MEng, PEng
Major (retired)

In Partial Fulfilment of the Requirements for the Degree of
Doctor of Philosophy

April 2014

This work is dedicated to two incredible people, my wife and my late father.

My wife, Dr. Catherine Louise Leblanc, has shown me what it means to persevere and to sacrifice; this work is for her as a token of the life we have built together – I would not change it for anything. You are my world.

Mon Père, le feu Paul-Émile Alcide Leblanc, outilleur-machiniste, est décédé en 2001 alors que j'entamais à peine le doctorat. Mon père a toujours supporté tous mes efforts; cet ouvrage est pour lui, comme preuve de ce qu'il aurait pu faire s'il avait eu les opportunités qu'il m'a aidé à obtenir.

ACKNOWLEDGMENTS

Many people deserve acknowledgements for this work. First and foremost are my family: my wife Catherine, my son Benjamin and my daughter Kate; the work is mine but the sacrifice is theirs in great part. Thank you for keeping me sane, for allowing me to keep going and for your unconditional love and support – without you, this would never have happened.

I want to thank my adviser, colleague and friend, Dr. Scott Knight, for sticking with me throughout this long endeavor. Scott is a tremendous researcher, and he is in very high demand. I am grateful for the time and attention he devoted to me; I know how many things I was up against and I thank you for the guidance you provided me. Thank you also for the encouragements, the support and for your uncompromising standards.

Dr. Ron Smith has been an island in a stormy sea. Ron's quiet and unassuming support has meant more to me than he probably knows. Thank you for the encouragement, for the true happiness at my successes, and for remembering what it feels like.

Thank you also to my colleagues, you have inspired me to keep going – I am not going anywhere.

ABSTRACT

The traditional response to the discovery of a network compromise has been to remove the compromised system from the network, to clean it and to restore it to service. This approach is reactive; a more active approach is necessary and such an approach requires better intelligence collection on attackers. New tools and techniques are required to allow the collection of intelligence on attackers, including the provision of realistic user activity at the human interface device (HID) level. This research developed a conceptual framework for the automatic generation of HID events in a manner that, when observed by attackers, is consistent with a human inputting text into a computer system. The framework, called the *Human Interface Device Event Generation Process*, accepts a target document as its input and, through sequential transformation, generates a series of mouse and keyboard human interface device events. When placed on the Universal Serial Bus of a compromised computer system, these human interface device events render the composition of the target document by a synthetic user. In order to make the generation of human interface device events consistent with what is expected of a human user, the framework makes use of a *User Personality Model* which represents the synthetic user's text composition preferences, editing choices, typing accuracy, use of the mouse and timing characteristics of human interface device events. All of these aspects of the User Personality Model are defined in this research. To demonstrate the validity and feasibility of the framework, we have developed a proof-of-concept *Synthetic User Environment* which implements the keyboard aspects of the *Human Interface Device Event Generation Process* framework. The research contributes to the field of computer network defence by providing a framework for the automatic generation of human interface device events, defining User Personality Model components and providing tools for the advancement of Network Counter-Surveillance Operations and Deception Operations.

RÉSUMÉ

La réponse typique à la découverte d'une compromission réseau est de retirer le système compromis du réseau, de le nettoyer et de le remettre en service. Cette approche est réactive; une approche plus active est nécessaire et une telle approche requière une meilleure cueillette de renseignement sur les attaquants. De nouveaux outils et techniques sont de mise pour permettre la cueillette de renseignements sur les attaquants, y compris la fourniture d'activité d'utilisateur au niveau de dispositifs d'interfaces humaines. Cette recherche a conçu un cadre conceptuel pour la génération automatisé d'évènements de dispositifs d'interfaces humaines de sorte à ce qu'ils soient consistants, lorsqu'observés par les attaquants, avec un humain composant du texte sur un système informatique. Le cadre conceptuel, appelé le *Human Interface Device Event Generation Process*, accepte un document-cible comme entrant et, de par une séquence de transformation, génère des évènements pour dispositifs d'interfaces humaines pour une souris et un clavier. Lorsque placé sur le bus USB d'un système informatique compromis, ces évènements pour dispositifs d'interfaces humaines simulent la composition du document-cible par un utilisateur synthétique. Afin que la génération d'évènements pour dispositifs d'interfaces humaines soit consistante avec un utilisateur humain, le cadre conceptuel utilise un modèle de personnalité humaine qui représente les aspects suivants de l'utilisateur synthétique : ses préférences de compositions, ses choix à titre de révision, son exactitude dactylographique, son utilisation de la souris et les délais associés avec les évènements pour dispositifs d'interfaces humaines. Tous ces aspects du modèle de personnalité humaine sont définis dans cette recherche. Une démonstration de la faisabilité, nommé le *Synthetic User Environment*, a été développée et mise en œuvre afin de démontrer la validité du cadre conceptuel. La recherche fait contribution à la défense de réseaux informatiques en fournissant un cadre conceptuel pour la génération automatique d'évènements pour dispositifs d'interfaces humaines, en définissant les aspects du modèle de personnalité humaine et en fournissant des outils pour l'avancement de opérations de contre-surveillance réseaux et les opérations de déception.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS AND ACCRONYMS

A	Absolute typing speed measure
Accident_But	Accidental Button
API	Application Programming Interface
ATA	Action Type Histogram
ATH	Action Type Histogram
AVS	Anti-Virus Software
CompError	Composition Error
Copy_SE	Copy Syntactic Element
Cut_SE	Cut Syntactic Element
DD	Drag-and-Drop Mouse Action
Delete_SE	Delete Syntactic Element
DND	Department of National Defence
DocMod	Document Model
DPM	Document Production Model
FAR	False Alarm Rate
GUI	Graphical User Interface
HCI	Human-Computer Interaction
HGP	HID Event Generation Process
HID	USB Human Interface Device
IDS	Intrusion Detection System
ii-CompErrorLex	Composition Error Lexicon
iii-GenEditActionLex	General Editing Actions Lexicon
IPR	Impostor Pass Rate
i-SynElmtLex	Synthetic Elements Lexicon
iv-SpecActionLex	Specific Editing Actions Lexicon
KOR	Keyboard Output Report
MDA	Average Movement Speed per Movement Direction
MDH	Movement Direction Histogram
Misplaced_SE	Misplaced Syntactic Element
Missed_Loc	Missed Location
Mistyped_SE	Mistyped Syntactic Element
MM	Mouse Movement
MOR	Mouse Output Report
Move_SE	Move Syntactic Element
MSD	Movement Speed compared to Distance Travelled
MTH	Movement Elapsed Time Duration Histogram
NCSO	Network Counter-Surveillance Operation

NIST	Network Intelligence Surveillance Toolset
OS	Operating System
Paste_SE	Paste Syntactic Element
PC	Personal Computer in general
PC	Point-and-Click in the context of mouse movement
PCI	*Peripheral Component Interconnect*
PDF	Probability Distribution Function
Position_Cursor	Position Cursor in Document Model
R	Relative typing speed measure
RCR	Remove-Clean-Restore
Replace_SE	Replace Syntactic Element
ROC	Receiver Operating Characteristic
Select_SE	Select Syntactic Element
SUE	Synthetic User Environment
TargetDocument	English free-text document for which the HGP generates keyboard and mouse HDI events.
TDH	Travelled Distance Histogram
Type_SE	Type Syntactic Element
UCS	Universal Character Set
USB	Universal Serial Bus
UTF-8	UCS Transformation Format – 8-bit
vi-HIDEventLex	HID Event Lexicon
vii-TimedEventLex	Timed Event Lexicon
v-MouseBehaviourLex	Mouse Behaviour Lexicon
WordProcMod	Word Processor Model
Wrong_But	Wrong Button
Wrong_SE	Wrong Syntactic Element

CHAPTER 1 : INTRODUCTION

The research described in this dissertation proposes a conceptual framework for the generation of user inputs on Universal Service Bus (USB) Human Interface Devices (HID) in order to facilitate intelligence collection on attackers.

1.1. Chapter Introduction

Removing a compromised computer system from a network upon discovery of a breach is not always possible because some key systems cannot be taken offline. It may also be undesirable because doing so would deprive the defender of the opportunity to collect intelligence on the attacker. Attackers who are aware of observation by defenders may alter their behaviour, thus invalidating the intelligence collected on them. Attackers are willing to spend significant effort to characterize the computer systems that they compromise in order to assess their value; the HID events present on the USB can be used by attackers to discover if the compromised system is used by a human or an automated process which could reveal the presence of a trap. This dissertation describes a conceptual framework for the automatic generation of USB HID events associated with the composition of arbitrary text; we call this framework the *HID event Generation Process*, or HGP. The approach was validated through the implementation of a proof-of-concept *Synthetic User Environment*, or SUE. This work is important because the proposed framework will help facilitate intelligence collection on attackers by generating HID events which are consistent with those generated by a human.

1.2. Response to Network Attacks

Because computer networks contain valuable information, they are subject to attack. These attacks can take many different forms, from broad-based scans by unsophisticated users to more serious attacks by very proficient individuals and organizations, such as foreign intelligence services, targeted at a specific organization. This section will discuss responses to network attacks, as done in [1].

1.2.1. Reactionary Response

The common computer security approach is to protect the systems as much as possible by employing vulnerability-oriented security measures (such as having up-to-date configuration of operating systems and applications, ensuring good security practises from users, using perimeter defence tools such as firewalls, etc.), detecting potential problems (by monitoring the network, using tools such as intrusion detection systems (IDS), etc.) and finally reacting to hostile events. Reaction too often takes the form of *Remove*, *Clean*, and *Restore* (RCR) [2], where defenders react by removing the infected system, cleaning it to remove the attackers' presence and restoring service. Unfortunately, there are three major problems with this RCR approach. First, there are many instances where the compromised system is critical to the operation of the network and simply cannot be removed from service. Second, because of the inherent complexities in computer systems and of the latent vulnerabilities in computer hardware and software, there is a high probability that such an approach will not adequately remove the attackers' presence. Too often, the computer security community attempts to protect information systems by building walls to stop potential attacks, but there is no one looking over these walls. Finally, the attacker has the initiative; the security community is only reacting to malicious activities and this makes it very difficult to keep abreast of new and evolving threats.

Network defenders need a better approach that allows them to learn about the attacker, as discussed by Spitzner who explains why it is necessary to understand attackers in order to be able to effectively defend against attacks [3]. In his work on honeypots, he suggests that an effective way to gain knowledge about attackers' tools and techniques is to place attractive targets for them to attack, and to observe the interaction between attackers and the target. The information gained from the interactions can be interpreted to discover which known vulnerabilities are exploited by the attackers, or even lead to the discovery of new vulnerabilities, which can in turn result in better protection.

Throughout Spitzner's discourse on the topic, the protection process always starts with a successful or attempted intrusion by attackers; the attackers always have the initiative.

1.2.2. Seizing the Initiative

There are risks associated with the traditional RCR response, which advocates the eradication of the threat by cutting off the attackers' access to the compromised system. These risks include the inability to identify the attackers; the loss of opportunity to learn about the attackers' techniques and motivations; the loss of ability to influence the attackers' actions. Finally, it can be argued that by cutting off the attackers' access where we know he has penetrated network defences, we are only encouraging intrusion into the network through other ingress points.

It is not sufficient to learn about attackers' tools and techniques; it is necessary to gather intelligence about the attackers. An intrusion into a computer network raises many questions. Who is attacking? What are the attackers capable of? What are their short-term objectives (in others words what are they after with this specific attack)? What are the attackers' larger strategic objectives and how does the current attack fit into that context? These questions cannot be answered if we simply deny the attackers access whenever an intrusion is observed. To answer these and other important questions, it is necessary to maintain contact with the attackers. This argument for the maintenance of contact is at the core of the motivation for the proposed research. The argument has been tested through peer-reviewed publication [4] [5] [6].

Leblanc and Knight [5] propose network counter-surveillance operations (NCSO) as a response to network attacks which have the potential to wrestle some of the initiative away from the attackers. This response is predicated on three operational objectives.

1. *Holding contact with the attacker*. Holding contact enables defenders to gain intelligence on attackers through their continued interaction with the compromised system.
2. *Understanding the attacker*. If we can gain an understanding of the attackers' capabilities, motivations and goals, we can be freed from a purely reactive response to intrusions.
3. *Preparing*[1] *the attacker for future operations*. An organization with the mandate and resources to take the fight to the attacker may launch a counter-operation on the attacker's information elements or infrastructure, or it may gather evidence for legal proceedings.

NCSOs are a significant research thrust of the RMC Computer Security Laboratory, and the associated statement of deficiency will be discussed in section 1.5. For the moment, it suffices to say that because it is predicated on holding contact with the attacker, there are risks associated with an NCSO. The attackers can damage or alter information on the compromised system; defenders may not be able to prevent the attackers from exfiltrating sensitive information; the attackers have more opportunities for pushing the attack onto other systems in the defended network.

Remaining undetected is of significant concern for this research. Should the attackers realize that they are being observed on the compromised system, they may alter their behaviour and thus make the intelligence collected by defenders less valuable.

1.3. Threat Model

[1] *Preparation* is used in a military intelligence context where it means taking actions to increase the probability of success of future operations. For example, a military organization planning a Deception Operation where they want their enemy to believe that a major air offensive is planned for a certain sector, may prepare said enemy by increasing air-reconnaissance activities in that sector.

To gain a shared understanding of the aim of this research, in order to demonstrate its validity, it is important to properly characterize the sophisticated attackers that are the subject of this research. There are many threats to any networked computer system, particularly if that system is connected to the Internet; computers are routinely scanned for the presence of known vulnerabilities. Many commercial products, such as anti-virus software (AVS), analyse these broad based threats to derive signatures. This research, in contrast, deals with very sophisticated attackers such as foreign governments' intelligence services or military forces. These attackers are likely to be well resourced and they have a strong interest in ensuring that their activities are not discovered, as is evident in the dearth of open publication on their exploits. The sophisticated attackers with which this research is concerned have some basic capabilities (see Figure 1 - Attacker Characterization), which they may use to characterize user activity on a compromised system.

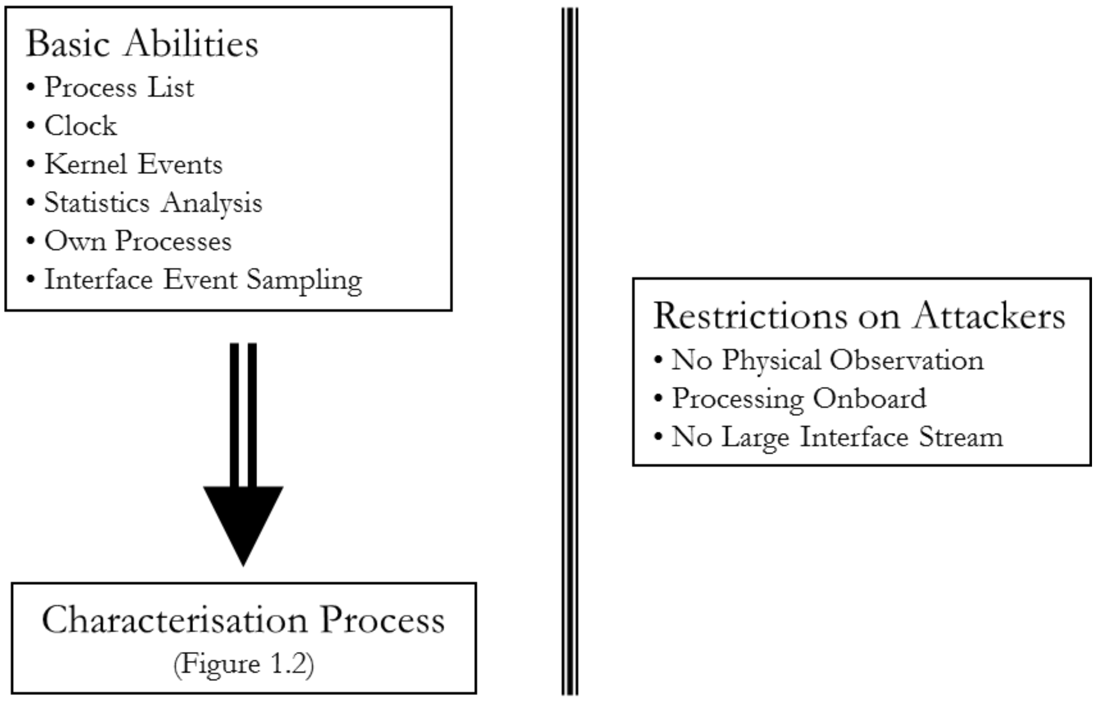


Figure 1 - Attacker Characterization

As we will discuss in Chapter 2, user characterization is one of the means available to attackers in order to ascertain the value of a compromised system. However, in order to continue deriving benefit from the compromised system, attackers wish to remain undetected and this places restrictions on how they can characterize user activity. The abilities of the attackers and the restrictions with which they can exercise those abilities are explained in this section.

1.3.1. Attacker Characteristics

We must presume that the compromise of the system is complete in that the attackers have administrative or root privilege on the target, which means that the list of processes running on the compromised system is visible to them. The attackers have access to the system clock, which will allow them to conduct statistical timing analysis of observed events. The attackers can observe events in the operating system (OS) kernel, be they in user-space or in kernel space. Along with visibility inside the OS, we will presume that the attackers can also run their own processes on the compromised system.

However, there will be restrictions on the attackers' abilities to characterize user activity, mostly because of a desire to remain undetected, which will result in the process depicted at Figure 2. The attackers do not have access to the system's physical environment, be it through direct observation or remotely through the use of surveillance devices. Without physical observation, attackers are therefore unable to see whether a user is actually using the HID interface devices. The sophisticated attackers that concern us will wish to avoid being detected in order to keep deriving benefit from the compromised system. Because of the standard reactionary response to intrusion discussed in section 1.2.1, the attackers must presume that they will lose access if their compromise is detected. The attackers will wish to process information onboard the compromised system, where they

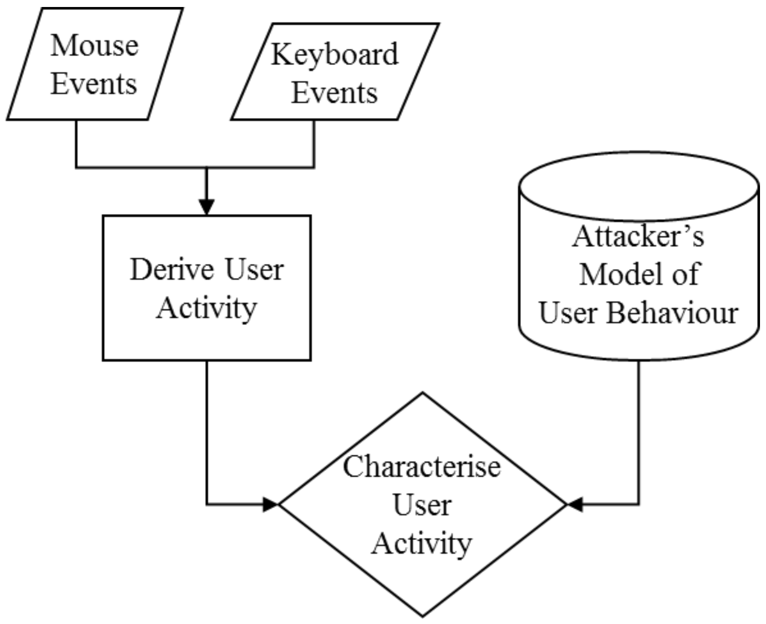

Mouse Events
Keyboard Events
Derive User Activity
Attacker's Model of User Behaviour
Characterise User Activity

Figure 2 - Characterization of User Activity by the Attackers

have a better chance to hide their activity, rather than exfiltrate large amounts of information off the compromised system because of the associated increase in probability of detection when transiting the network. Although attackers can sample events associated with HID, as discussed in section 1.3.3, the attackers will not be able to stream all HID interface events to recreate user activity because such streaming would use significant network bandwidth, and would increase the probability that their attack will detected by an IDS [7].

1.3.2. Characterization of Users by Attackers

To characterize the user of a compromised system, attackers can observe system kernel events associated with HID, specifically mouse and keyboard activity. For each mouse or keyboard event, the attackers will be in a position to observe the type of event and its time of occurrence. From these observed events, the attackers will be able to derive a limited representation of the user activity. It is reasonable to presume that attackers will have models of user behaviour at that HID interface level, as some of these have been published in the open literature (see section 2.4). Attackers will try to characterize a user by testing the activity derived from observed user inputs against their known models of user behaviours.

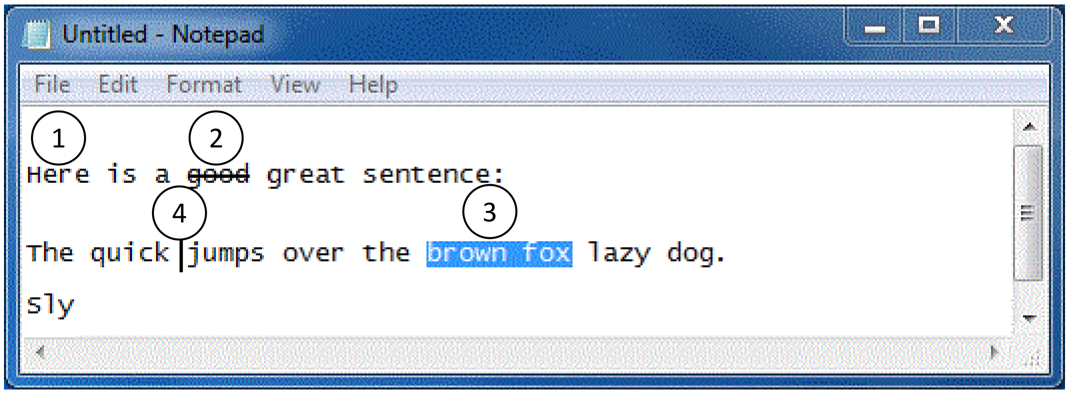
1.3.3. HID interface Event Stream Semantics

We can think of the HID interface event streams as a language with specific syntax and semantics. Defining the syntax of the HID interface event stream is a bounded problem, where it is possible to define the tokens (the characteristics of the keyboard and mouse events) of the language. It is therefore reasonable for the attackers to compare the user activity derived from observing the USB to models of behaviour using statistical testing. This approach would be particularly well suited to processing of the HID interface event streams on the compromised system itself, during quiet hours, when the system's legitimate user would be less likely to detect it.

The user moves the mouse and types at the keyboard to accomplish tasks. The semantics of the HID interface event stream is the meaning associated with the mouse and keyboard events. There have been many research efforts aimed at characterizing natural languages using formal mathematical notations such as [8]. It is important not to confuse these with this research because the former is premised on an analysis that has access to much more resources than the attacker described in the latter.

One can think of the simple task of drafting an email message to understand how quickly the HID interface event stream becomes complicated (Figure 3 helps understand this example). In this example, the cursor is moved in the typing window (1), and the user starts typing. A mistake is made, causing the user to use the backspace key to erase some text, as represented by the strike through at (2), before correcting it. Perhaps a portion of a sentence needs to be moved; it must be selected (3) and cut (using the keyboard, the mouse or both) before the mouse is used to reposition the cursor (4) in order to past the text (once again using the keyboard and/or the mouse).

The attackers would be able to make a judgement about the semantics of the document creation task through direct observation of the user, or by exfiltrating the HID interface event stream and reconstructing it. As we discussed in section 1.3, the attackers' desire to remain undetected will preclude them from being able to exfiltrate the entire HID interface event stream as this raises the probability they will be detected. The attackers would be limited to exfiltrating a group of HID interface events to see they represent syntactic text elements arranged in a hierarchy of words, numbers, and punctuations to form sentences.



```
Here is a ~~good~~ great sentence:

The quick |jumps over the brown fox lazy dog.

sly
```

Figure 3 - Event Stream Semantics

We will refer to the information gained from such reconstructions through event stream sampling as *local semantics*. Local semantics are in contrast to global semantics, where more general meaning is derived about the document creation activity. An example of local semantics would be to say that the attackers can derive the word `the` from the character sequence `teh [backspace] [backspace] he`. Global semantics on the other hand, would allow the attacker to realize that the conclusion of a report was drafted before the introduction.

The attackers may well choose to create a tool to examine the event stream on the compromised system itself. Such a tool would allow the attackers to communicate less information using the network than they would if they were to exfiltrate the entire event stream, thereby reducing the probability that their compromise would be detected [5]. Such a tool would have the ability to use local semantic information to analyze the event stream.

In summary, we can say that a realistic HID interface level environment is necessary to prevent attackers from changing their behaviour, which would hinder intelligence collation. Although there are a number of potential solutions for providing such a realistic HID interface level environment, this research focused on the automatic generation of user activity, as it is deemed to be cost effective. The proposed automatic generation of user activity is also realistic in the context of a threat model where the basic abilities of the attackers are restricted by their motivation to remain undetected. Under these restrictions, the attackers will be able to use their basic abilities to characterize user activity by comparing mouse and keyboard events to known models of user behaviour (see section 2.4). The goal is for the attackers to characterize the automatic user activity generated by this research as human activity. Such automatically generated activity will be deemed consistent with human behaviour because it will respect the syntax of the English language and local semantics of the event stream defined in the models of user activity that are at their disposal.

1.4. Operational Scenario

An organization may decide that the risks associated with an NCSO, as discussed at the end of section 1.2.2, are justifiable when compared to the potential benefits of the collected intelligence. When this occurs, the organization may decide to engage a Network Intrusion Surveillance Toolset (NIST)[2] to begin an intelligence gathering operation. Many parallels can be drawn between the NIST and honeypots, which will be discussed in detail in section 2.2. The NIST will require three categories of tools, namely 1) tools to surreptitiously observe attackers, 2) tools to limit the potential damage that can be done by attackers while on the compromised system and 3) tools that will keep attackers engaged with the compromised system.

This research is particularly interested in the latter. For the intelligence gathering operation to be successful, the attackers must be convinced that the system they have compromised is a legitimate, high-value asset. This research presents a framework for the generation of HID events and implements a proof-of-concept of this framework in a SUE that will mimic user activity at keyboard HID interface level, in order to convince the attackers to continue interacting with the compromised system, thereby allowing intelligence collection on them.

[2] The NIST is a research thrust of the RMC Computer Security Laboratory.

This is best illustrated with a representative scenario, which we will situate in a classified network of the Department of National Defence (DND). This network is used in support of national command and control, and it contains classified information. The network is protected by perimeter network security devices such as firewalls and IDS, but in this scenario, it is DND's policy to allow limited connectivity to the Internet (possibly only outbound HTTP connections) to enable network users more flexibility in the performance of their duties[3]. Imagine that sophisticated attackers have been able to target, attack, gain access to and compromise a system residing on that network; let us further presume that it is a workstation used by a senior commander's secretary. The presence of that system on the classified network highlights its value to DND, as does the presence of network protection measures such as firewalls and IDS. The system is also very valuable to the attackers for two reasons: first, it may contain valuable information because of its close association to a senior commander, second it is a presence on the classified network from which the attackers can carry out a broader range of network operations.

This scenario represents a situation where the NCSO described by Leblanc and Knight [5] may well apply; it is conceivable that the potential risks to which DND is exposed by maintaining contact are deemed justifiable when compared to the potential intelligence benefits derived from continued contact with the attackers. In order to respond to the attack in a manner that would allow defenders to carry out the three operational objectives stated in section 1.2.2, it is important that the attackers believe that they are interacting with a legitimate system, and not with a decoy or honeypot. Sophisticated attackers however, are not fooled easily and they can discover that they are dealing with a suspicious computer system (see Vitality Detection in section 2.4). The consequences of the discovery by the attackers that they are being observed can be detrimental to DND. At the very least, the attackers may stop interacting with the compromised target, which means that their behaviour can no longer be observed. Of more concern is the fact that the attackers may retaliate by damaging the compromised system or other systems on the classified network or by feeding erroneous information to those attempting to observe their behaviour. In other words, detection of the attackers will likely cause them to alter their behaviour, which may invalidate any collected intelligence.

[3] Such a connection to the Internet is not necessarily present in any actual secure network, and the author does not advocate its use. It is only suggested here to illustrate an obvious entry point for attackers to gain access to the network.

In order to minimize the risk of detection of the defenders' observation by the attackers, the target must appear to be a legitimate system. For a system that is assigned to an individual, a secretary in the case of our scenario, this legitimacy includes user actions through HID. As the attackers have the ability to observe the interactions between HID and the operating system, the HID interfaces provide a potential means by which the attackers can characterize the system. It is thus important to the maintenance of contact that the compromised system appears to have legitimate user activity at the HID interfaces.

There are many potential ways of providing user interaction that appears realistic at the HID interface level. First, we may decide to leave the user, the senior commander's secretary in our scenario, interacting with the compromised system. This solution is very simple as it does not involve additional personnel or special technology. This approach is not always feasible however, because it restricts the user's ability to perform regular duties because of his/her involvement in the intelligence operation. The intelligence collection operations may also be classified at a level higher than the user's security clearance. Finally, concerns for the privacy of the user may also make this approach impractical.

Defenders could capture HID interface level user activity and replay it as a second approach to the generation of user activity at the HID interface level. This would ensure that the HID interface activity is statistically consistent with a user of the system, which is good since the attackers can use statistical analysis to characterize the system (as discussed in section 1.3.2). This would also prove to be inexpensive to implement. However, attackers who maintain state information about the compromised system could detect this approach by realizing that the same activity is taking place more than once. Such an approach is likely to not pass the local semantics requirements discussed in section 1.3.3. Further, replaying HID interface level user activity would likely not allow the defenders to prepare the attackers for some types of further operations, such as planting specific information on the compromised system for the attackers to find. This approach would thus fail the third operational objective described in section 1.2.2.

A third alternative would be to have intelligence operators or analysts generate HID interface level activity on the compromised system by interacting with it. This approach would clearly provide a realistic HID interface level environment, and it would allow defenders to enter arbitrary information on the compromised system, thus possibly preparing the attackers for further network operations. However, this approach would be prohibitively expensive. Intelligence operators and analysts are highly trained resources; they possess specialized skills and they have gone through an onerous security screening process. Because they can be employed in very sensitive operations and other specialized intelligence activities, one can surmise that their skills are highly valuable. While it appears that using an intelligence operator or analyst to simply interact with the compromised system's user interfaces is a very inefficient use of valuable resources, a comparative analysis of the costs falls outside the scope of the current research. We will nonetheless reject this approach as not feasible.

The fourth alternative being advocated by this research suggests a way to automate the generation of user activity at the HID interface level. The approach described in Chapter 3 is superior to the three alternatives discussed above because it does not involve the compromised system's regular user, or that of valuable intelligence operators or analysts, it does not violate the local semantics that the attackers could detect through sampling of the HID interface event stream, and it allows for the introduction of arbitrary text on the compromised system.

1.5. Statement of Deficiency

The reactionary RCR response to a compromised system inside a friendly network described in section 1.2.1 is deficient because it breaks contact with the attacker; in order to seize the initiative and gather useful intelligence, defenders must maintain contact with the attackers on the compromised system. We have advocated for the use of an NCSO as a means to gather intelligence, and we discussed the objectives that can be met by such an NCSO in section 1.2.2. This section aims to show how the research fits in the greater scope of NCSO.

Much work needs to be done to effectively employ NCSO. Organizations must accept that they are necessary, and new tools must be developed in order to carry them out, as detailed in previous publications [5] [6]. More specifically, the NIST tools must fulfil the following three requirements:

1. surreptitiously observe attackers actions on the compromised system,
2. limit the damage that attackers can cause from the compromised system, and
3. provide a realistic environment on the compromised system to keep the attackers engaged.

Observation of attackers on the compromised system is necessary to the collection of intelligence. Attackers are by definition non-cooperative, and will therefore not assist defenders in their observation efforts. In order to keep attackers from changing their behaviour, thereby invalidating collected intelligence, the observation must be made surreptitiously. Surreptitious observation is made difficult by the fact that attackers have administrative privileges on the compromised system, with the abilities discussed in section 1.3.1. Various techniques have been investigated to carry out surreptitious observation using virtual machine monitors [9] [10] and rootkit technologies [11], but much work remains to be done. The need to surreptitiously observe the attacker also affects how the other two NIST requirements can be fulfilled.

There are risks associated with the conduct of NCSOs. While the benefits of the collected intelligence can make the risks acceptable, efforts must be made to control the many forms of damage that attackers can cause using the compromised system. For example, attackers can use a compromised system as a foothold to carry out network reconnaissance or to prosecute attacks deeper into the network on which the compromised system resides. Serious liability or damage to the defenders’ reputation can also occur if attackers are able to use one’s compromised system as stepping-stone to attack other systems that reside on the networks of partners, friends or allies. There exists a trade-off between controlling the actions of the attackers and remaining undetected by them because the actions taken by defenders to control compromised systems can alert attackers that their activities have been discovered. This concept has been introduced in [5] and it remains an open area of research.

The third series of tools required by the NIST toolset aims to ensure that the compromised system appears realistic to the attackers. Attackers wish to derive benefits from the high value computer systems that they compromise; we surmise that they are therefore likely to explore them in detail in order to become familiar with their contents (be that files, accounts, installed applications, etc) and their operating environments such as what applications are installed, which processes are running, etc. It is not surprising that the research literature does not contain details of the ways in which attackers will characterize compromised computer systems, but we argue that any significant changes to the way in which the compromised system is used can tip off the attackers that they have been discovered. Recall from our Operational Scenario (section 1.4) that the NIST is activated after a system has been compromised, but none of its tools can be deployed on the compromised system after the compromise is discovered. In order to keep the attackers engaged, the compromised system environment cannot change to become unrealistic. For example, adding or removing applications or processes that render the compromised system inoperative after the compromise are likely to tip-off the attacker that they are under observation.

1.6. Aim

While the attackers’ characterization of the compromised system can take many forms, we argue that it may include the monitoring of USB HID. Indeed, attackers who compromise a user workstation such as the one described in our Operational

Scenario would easily notice if the USB HID activity stopped on the compromised system. The automatic generation of USB HID events on a computer system is therefore important, and the automatic generation of mouse and keyboard HID events is the specific problem tackled by this research.

The aim of the research is to develop a conceptual framework for the automatic generation of HID events in a manner that, when observed by attackers, is consistent with a human inputting text into a computer system.

A readily available means of generating USB HID events would be to simply record them, and replay them on the USB of the compromised system under observation; this approach is not suitable to meet the needs of NCSOs. The attackers against whom we wish to collect intelligence are deemed to be sophisticated and highly motivated. Such attackers are not easy to fool, which would make it likely that the replay of HID events on the USB would be detected. There is therefore a need to present the attacker with HID activity on the USB that is consistent with HID events generated by a human inputting text into the compromised system[4]. The operational scenario presents a hypothetical situation where the maintenance of contact with the attackers is desired, but it also highlighted the opportunity to use the compromised system as a means to feed certain desired information to the attacker. This research therefore proposes a means to generate HID events that are associated with the input of arbitrary text into the compromised computer by a synthetic user.

The conceptual framework resulting from this research encompasses an HID Event Generation Process (HGP) which produces a series of HID events corresponding with the inputting of text into a computer system. The HGP uses a *pipes and filters* architecture which accepts a target document containing arbitrary text as its input, and produces a series of HID events along with inter-HID event delays, which corresponds to the inputting of that target document on the compromised computer system.

In the context of this research, a *conceptual framework* is taken to be the definition of the inputs and outputs of each stages of the HGP, along with the lexicons that are used by the stages. The conceptual framework also provides a design of each of the transformations that implement these stages, and descriptions of the use of models of user personality.

[4] We refer to the notional user which the framework emulates as the *Synthetic User* for the remainder of this dissertation.

The automatically generated HID events will be considered consistent with those generated by a human inputting text if the attacker is unable to distinguish the former from the latter as shown at Figure 2. Recall from section 1.3 that the attackers have some capabilities on the compromised system, but that their desire to remain unobserved imposes limitations on what they can do.

In order to produce HID events that are consistent with a human user inputting text into a computer system, the HGP utilises models of user behaviours. These models detail the HID proficiency of the user, therefore allowing the framework to generate HID events representing errors associated with the use of HID such as typing mistakes or mouse movement errors. Because there are often many different but equivalent ways to edit a document while inputting text, the user models also represent the user's preferences in terms of editing actions. Attackers have the ability to stream portions of the HID event stream (Section 1.3.3); the HGP therefore also includes composition errors that ensure that the generated HID event stream is consistent with a human user composing text as it is inputted into the compromised system. The various models of user behaviour allow the HGP to be parameterised to represent different synthetic users in a consistent manner.

The HGP describes the generation of HID events for a mouse and a keyboard. While there are many different HID, we believe that the mouse and keyboard are representative of the major activities involved in the inputting of text, namely typing, pointing and selecting. The pipes and filters architecture chosen to implement the proof-of-concept application can be extended to include the use of other HID and we deem the use of these two HID devices sufficient to demonstrate that the automatic generation of HID events is possible.

Text is entered in a computer system through an application. There is a wide variety of applications used for this purpose, but many share a certain number of characteristics. Each of them has a text entry area that can be thought of as a grid of columns and rows, and each has commands used to carry out various actions. We chose to use Microsoft Notepad in the implementation of the proof-of-concept HID Event Generation SUE application. We believe that this is sufficient to demonstrate the feasibility of the proposed framework.

The automatic generation of HID events is necessary to maintain contact with attackers who have compromised computer systems. While this will likely not be required widely in the defence of computer networks, it is very important for NCSO. While the deployment of NCSO is not likely to be widespread, they are required in limited but very serious situations where network defenders require intelligence on attackers [6].

This research has developed a complete HGP pipeline that takes arbitrary text as its input, and generates a sequence of mouse and keyboard HID events that are consistent with a human user composing text when placed on the USB. Each stage of the pipeline has been designed, along with the models on which the pipeline depends.

1.7. Validation Approach

To the best of our knowledge, current literature does not detail any research into the generation of the HID events associated with the production of arbitrary text in a manner that is consistent with a human using a plain-text editor. While this gives an indication of the originality of the work, it also means that it will not be possible to validate the proposed HID event generation framework against other known works. The validation of the work will therefore be done argumentatively through the structured reasoning that is discussed in this section.

We have briefly discussed why we believe that the problems associated with the maintenance of contact with attackers on computer networks are important to NCSOs; Chapter 2 – *Literature Review* further demonstrates that this problem is real and valid. The chapter will discuss the emergent field of Cyber Deception to show that this is a promising area of research. Honeypots and Honeynets have been proposed as means to learn about attackers, and the chapter will discuss those and demonstrate why they are insufficient for the NCSOs discussed in section 1.2.2. The chapter also discusses how attackers are willing to spend considerable time, energy and resources into the characterization of the computer systems that they compromise by a discussion of *vitality detection*, which form the basis for three of the five models of User Personality.

Having argued that the problem identified in section 1.6 is real and valid, Chapter 3 – *HID Event Generation Process* presents the proposed framework. The framework takes as its input a *TargetDocument* and it produces the HID events and inter-HID event delays corresponding to a synthetic user entering this text in the compromised system through the following stages:

1. Extraction of syntactic elements such as words, numbers, sentences, paragraphs, etc from the *TargetDocument* to populate a *Document Production Model* (DPM)
2. Introduction of *Composition Errors* in a DPM according to the *Composition Model and Typing Accuracy Model*
3. Modification of the DPM to represent the selection of *Editing Actions* in accordance with the *Editing Model*

4. Augmentation of the DPM to represent the addition HID events which represent the use of the keyboard and mouse in accordance with the *Editing Model*
5. Introduction of *Mouse Errors* in the DPM in accordance with the *Mouse Accuracy Model*
6. The generation of HID Events, namely *Keyboard Output Reports* and *Mouse Output Reports*, which can be placed on the USB of the compromised system
7. Introduction of inter-HID event delays in accordance with the *HID Timing Model*

Because the HID events automatically generated by the synthetic user must be consistent with those entered by a human user, the chapter describes the models of user personality that are used to introduce errors in the synthetic user composition task, as mentioned in the stages above. We argue that the proposed framework proposes a realistic environment that is not likely to bring the attackers to alter their behaviour on the compromised system, and thus makes the collection of intelligence on them possible.

Following Chapter 3, argues that the proposed framework is a valid solution to the real problem described by the research aim. Chapter 4 – Proof of Concept SUE, discusses the proof-of-concept SUE application which accepts arbitrary English free-text and produces a series of HID events that represent the composition of a *TargetDocument* on a compromised system running Microsoft Notepad. The SUE implements a thread of execution through the HGP to demonstrate the feasibility of the HGP. The SUE implements the aspects of the HGP that deal with keyboard events, but it does not implement those dealing with mouse events. Aspects of every stage of the process have therefore been implemented, except for Stage 5 that deals strictly with mouse events.

While it would have been interesting to implement every aspect of the HGP in the SUE, we chose to omit some because of equipment limitations and because we argue that they are not required to demonstrate feasibility. Chapter 4 presents our proof-of-concept requirements, gives the details of the implementation of the SUE and provides detailed arguments explaining why we believe that what was implemented was sufficient to demonstrate the validity of the HGP.

Chapter 5 – Summary and Conclusion concludes this dissertation by highlighting the shortcomings of other research efforts, namely in the areas of network intelligence collection and honeypots. It also discusses why we believe that the research aim has been met by the proposed framework and why the proof-of-concept SUE application demonstrates that the approach is valid. The chapter also highlights/ the unique contributions of the research and it will discuss avenues of future work.

CHAPTER 2 : LITERATURE REVIEW

2.1. Chapter introduction

We have established the need for collecting intelligence on attackers when an organization decides to maintain contact with them on a compromised system as is the case in NCSO throughout Chapter 1. It is well established that attackers and defenders are engaged in a sort of electronics arms race. Defenders discover some of the vulnerabilities that are being exploited and deny attackers their use; in response, attackers move on to exploit other vulnerabilities and the cycle continues. A similar cycle takes place in terms of intelligence collection and disinformation. Defenders want to collect information about attackers and attackers are willing to go through significant efforts to ensure that they are not observed.

In this chapter, we will look at honeypots as a means of collecting information on attackers, and discuss why they fall short of meeting our research aim. We then examine literature on *Cyber Deception* to demonstrate that it represents a significant area of research in computer security and argue that this research contributes to this field. The last section of this chapter will examine how attackers may characterize a computer system in order to determine the presence of a human user, which we call *Vitality Detection*, and which forms the basis for some of the Models of User Personality discussed later.

2.2. Honeypots

We have previously discussed the need to give attackers a realistic target with which they can interact. The idea of watching attackers while they are penetrating systems is not new; it was suggested as early as 1989 by Clifford Stoll in a book entitled The Cuckoo’s Egg [12], and Bill Cheswick’s paper *An Evening with Berferd* [13]. The premise is that if it is possible to examine attackers at work, defenders may be able to learn about their motivations and techniques. This in turn can provide useful information for designing protection systems and procedures.

This section will discuss honeypots as a means of creating such a target. We will discuss the premise and definition of honeypots, various classifications schemes for them, and conclude with a discussion of attackers’ efforts to detect honeypots.

Honeypots are applicable to this research, as important parallels can be made between them and the HGP framework described therein. Like the framework, honeypots help learn about attackers. Because honeypots are deployed on many networks, the research community has had opportunity to observe how attackers react to them; from this, important parallels will be inferred about the potential reaction of attackers to the framework. The section will conclude with a summary of the shortcomings of honeypot research in addressing the proposed research problem.

2.2.1. Premise and Definition of Honeypots

It is possible to learn a great deal from observing the activity of attackers on an organization's actual production systems[5]. Systems that access large networks such as the Internet are constantly connecting to other computer systems. Many of these connections are legitimately initiated by the user, or by processes acting on the user's behalf, but many of these are the results of scans initiated by other systems [3]. It can be extremely difficult to distinguish between genuine use by the production system's user and potentially harmful activity by attackers. Consider a simple example dealing with the `http` protocol. A typical Web page accessed by a browser can contain many different links to a wide variety of servers. When users access an on-line news service (such as `www.cbc.ca/news`) they will be seeing hypertext from the CBC, but they will also likely see advertisements from sponsors. These advertisements typically originate with a third-party organization, and not from the news agency. Most production systems also have a wide variety of active processes that permit many conveniences such as email, network file services, print services, etc. Many of these processes communicate with other systems on the network, generating traffic. When observing a compromised system, the attack activity becomes obfuscated by the legitimate use of the production system, and is thus difficult to detect, isolate and analyze. The security analyst is condemned to finding the proverbial needle in a haystack.

Enabling the observation of the attackers' activities is the premise behind the concept of honeypots. A honeypot is defined as *a security resource whose value lies in being probed, attacked or compromised* [3]. In this definition, a honeypot has no production value. The system should not be requesting any services from servers on the network and it should not offer any services to clients. Discovering attack traffic then becomes trivial because any connection or connection attempt to or from the honeypot is suspicious in nature.

Honeypots have been credited with many successes in detecting attacks and observing attacker behaviour. They have been used to capture information to track down spammers that send unsolicited emails [14] [15], to identify and track

[5] Production Systems are taken to mean those computer systems that are used by the organization for its core function. They are not specifically designed to learn about attackers.

individuals that defaced Web sites, and to discover a zero-day[6] Distributed Denial of Service Tool [16]. In an extensive study at the Georgia Institute of Technology, honeypots were used to capture the exploitation pattern of Internet worms by collecting data on how they propagated across the network. The information captured by the honeypots also led to the discovery of many infected systems on the local network, and they were useful in discovering poor computer security practises by some network users [17]. This is a short sample of exploits discovered by honeypot technology; the reader is referred to the following sources for a more extensive treatise on the topic in the field of web exploits [18], botnet detection and analysis [19] and wormhole detection [20].

2.2.2. Honeypot Risks

There are two major risks associated with the use of honeypots: being used as a stepping-stone, or being discovered. The honeypot can be used as a launching point (a so called *stepping stone*) for an attack on another system. This can be particularly damaging to the organization hosting the honeypot, as the honeypot may have better access to other systems inside the organization's security enclave. The organization hosting the honeypot must therefore be concerned with the risk to its own production systems, with damages to its reputation with allies and partners, and with legal liability resulting from damages to other systems launched from the honeypot.

There is also a risk that the attackers will discover that they are on a honeypot, and not on a production system. As we will discuss in section 2.4, attackers will go to great lengths to characterize the systems that they have compromised to discover if they are legitimate production systems. Attackers that discover that the system they have compromised is not legitimate have many options: they can stop interacting with the compromised system, they can retaliate by causing damage to the compromised system or other systems belonging to the organization hosting it, or they can mount a disinformation operation, by supplying erroneous information to the defenders observing their actions. This third response by the attacker is the worst from the perspective of this research, as it invalidates the defenders' observation of the attackers and wastes resources.

2.2.3. Honeypot Classification

Before an organization deploys honeypots, it must have a clear purpose for their use. The preceding paragraphs have given some examples of their benefit, but it is

[6] Zero-day is used to describe an exploit that was as yet undiscovered by the computer security community.

worthwhile to have a more thorough look at their value. Spitzner suggests that there are two types: production and research honeypots [3]. Production honeypots should not be confused with production systems, which are central to an organization’s core business or operational functions. Production honeypots are there to help secure the production systems. They are used to detect attacks and to detract attention from production systems. Spitzner draws a parallel to law enforcement: production honeypots are there to catch the bad guys so that the organization can deal with the immediate threat. It is usually not necessary to give attackers a lot of functionality when we only want to catch them, so production honeypots tend to be easy to deploy and many production honeypots come as easily installed applications. Research honeypots on the other hand are designed to gain information about attackers. In order to gain information about the attacker community writ large, the researcher may be willing to sacrifice some short-term security objectives such as patching every hole in the defence perimeter. In order to be able to gather intelligence, attackers must be given a reasonably realistic system to interact with. This tends to make the deployment of research honeypots more resource intensive.

While it is true that production honeypots have had success in helping to protect organizations, they are not the focus of this work. This work is aimed at gathering intelligence on attackers, it is therefore necessary to examine ways of gaining knowledge about specific threats. Responses to these types of threats have not been well documented in the open literature, and different kinds of tools will be required to properly address them. As discussed in section 1.5, this research investigates tools that can be used to turn a compromised system into a tool to collect intelligence data about the attacker. While such tools are similar in some ways to honeypots, they do not fit the definition of either production or research honeypot; it would more appropriately be defined as an intelligence-gathering tool.

Along with making a distinction between production and research honeypots, Spitzner also proposes a taxonomy that is based on the functionality given to the attackers once they have gained access to the honeypot [3]; such a classification may also be useful with regard to this research. A honeypot may be classified as *low*, *medium*, or *high* interaction, depending on how much the attackers are allowed to do with the system. In order to achieve a higher level of attacker interaction, while still maintaining control over what the attackers can do with the honeypot, more effort will have to be devoted to the honeypot configuration, deployment and maintenance. As we discussed in section 2.2.2, using honeypots can be risky. We must keep in mind that the more functionality the attacker is given to interact with the honeypot, the greater the risk. Table 2.1, taken from Spitzner, summarizes the trade-offs between the level of interaction afforded the attacker, the work required to install, configure, deploy and maintain the honeypot, the potential information gathering ability of the honeypot, and level of risk associated with its use.

Table 2-1 Tradeoffs between Honeypot Interaction Levels [3]

	Work to Install and Configure	Work to Deploy and Maintain	Information Gathering Potential	Level of Risk
Low	Easy	Easy	Limited	Low
Medium	Involved	Involved	Variable	Medium
High	Difficult	Difficult	Extensive	High

Low-Interaction Honeypots

Low interaction honeypots are easy to install and configure. In fact many ready-made solutions such as NFR Security's Backofficer Friendly (now rolled into Checkpoint Security's threat prevention appliances [21]), and others come as pre-packaged applications. Instead of mimicking a complete operating system, the honeypot only emulates a few specific services, and attackers are given limited interaction with those. For example, the honeypot may be emulating an FTP server by giving attackers a login prompt and capturing the account name and password used to attempt access along with other details about the login session. The honeypot could also emulate a file transfer service, allowing the attacker to download files that have no production value for the organization.

The main purpose of low-interaction honeypots is detection, and they are particularly useful at detecting network scans and unauthorized access attempts. Because they look for well-defined attacks, low-interaction honeypots are very efficient at recognizing known threats and they can be easily configured to alert system administrators when such attacks are detected.

Low-interaction commercial solutions are stable and usually require little work to set-up, deploy or keep in operation. Because they are only looking for known attack patterns, it is usually not necessary to give attackers a lot of functionality. This allows the organization to limit the potential damage that can be done by the honeypot by putting in place some very strict data control mechanisms. If data control is well executed, it can greatly limit the damage that can be wrought by a low-interaction honeypot.

Low-interaction honeypots require less work and represent less risk than medium or high-interaction honeypots. However, because they only look for known attack patterns, low-interaction honeypots are not very useful for discovering new attacks or novel ways of carrying out known attacks.

Medium Interaction Honeypots

Medium-interaction honeypots attempt to emulate services with more fidelity than low-interaction honeypots, while still not deploying a full version of the services being presented. Spitzner gives the example of a medium-interaction honeypot deployed to gain information about an Internet worm targeted at a Microsoft IIS server. While a low-interaction honeypot might only log connection information from the attacking system, a medium-interaction honeypot would establish the connection in a way that is consistent with the emulated IIS server. It is hoped that if the honeypot is configured properly, the worm will upload its payload following what it deems to be a successful connection, and the honeypot will be able to capture it. Because of the limited functionality given to the attacker, user-mode servers such as `chroot` and `jail` would typically be categorized as medium-interaction honeypots in Spitzner's taxonomy.

This increase in the potential information collected makes the medium-interaction honeypot useful in a more active network defence posture. In the previous example the worm was allowed a connection with the honeypot's operating system, but only through the emulated service. Because the worm has not connected to the actual service it was targeting, the payload will not have its desired effect. While this offers some protection, such a medium-interaction honeypot is riskier than a low-interaction honeypot, and requires significantly more effort to deploy and maintain. The gain in the information that can be potentially collected by the honeypot may be offset by the extra work required to keep the honeypot from becoming a risk to other systems.

High-Interaction Honeypots

Spitzner argues that the best way to observe attackers at work (and thus gain the maximum amount of useful information) is to give them a complete system with which to interact, and this is precisely what takes place in high-interaction honeypots whose goal is to give attackers access to a real OS where nothing is emulated or restricted [3]. A high-interaction honeypot is very often configured in an attempt to appear indistinguishable from the production systems deployed on the network, except that it has no production value. The high-interaction honeypot affords the most comprehensive opportunity for learning about the attacker's techniques, because it allows the attacker to interact with an entire operating system. However, this means that attackers have many resources with which to cause damage to an organization's networks. Attackers may also be able to use the resources of the compromised high-interaction honeypot to launch attacks against other targets, exposing the organization hosting it to potential liability or costing it good will with friends, partners or allies.

To mitigate the potential damage done by attackers, and therefore reduce the risk to the organization hosting the honeypot, much effort must be expended to control the compromised honeypot. The honeypot will therefore usually be deployed in a controlled environment, such as behind a firewall in a specially configured network zone with other assets having a similar security policy [22]. The firewall will be configured to allow the attackers to reach the honeypot to compromise it, but will restrict their ability to use it to attack other systems. It may completely prohibit connection attempts emanating from the honeypot, or it may use techniques such as limiting the bandwidth leaving the honeypot or introducing delays. The configuration of such a firewall can be complex and requires significant time and effort. As was illustrated in Table 2-1, high-interaction honeypots have the greatest potential for gathering information about attackers, are the most difficult to install, configure, deploy and maintain, and represent the greatest risk to the organization.

2.2.4. Applicability and Shortcomings of Honeypots to NCSO Research

The classification scheme proposed by Spitzner shows a hierarchy of categories which focuses on the level of interaction afforded to attackers with high-interaction honeypots at the top. The classification scheme recognizes that there is a correlation between the amount of effort devoted to the set-up and administration of the honeypot and the value of the potential information gained from it. Spitzner also discusses the trade-off between the level of interaction (and by extension the potential value of the information) and the risk to the organization. As was discussed in section1.2, we believe that there is some value in maintaining contact with attackers in order to gather intelligence with respect to their identity and motives. Honeypot technology has been used primarily to learn about attackers' tools and techniques; we believe however, that it is still useful to examine the HGP described in this research in terms of Spitzner' classification scheme.

Recall the scenario used to motivate the current research from section 1.4. Such a system would be subject to the organization's maintenance policy, and it would benefit from the organization's patch and update cycle. While it is possible that the compromised system on which an application implementing the HGP is deployed may not be perfectly maintained, it would be as up to date and secure as any of the other production systems; the system is not purposely kept insecure to attract attackers as is the case with weakened systems. In that context, the deployed application is similar to Brenton's hardened system [16].

While honeypots have contributed to computer security, we argue that they are not sufficient for the gathering of intelligence on attackers for three reasons:

1) NCSOs are deployed on systems with production value,
2) honeypots do not provide a realistic environment and
3) honeypots cannot prepare attackers for future operations. All three of these reasons have been deemed necessary in our discussion of deficiencies (section 1.5).

In section 2.2.1, we have stated that a honeypot has been defined as a system that has no production value; this fact bears repeating because it represents the first shortcomings of honeypots in the prosecution of NCSOs. The HGP is meant to be activated on a production system after it has been compromised, and this is necessary to maintain the attackers' interest. We argue that it will not be possible to maintain the contact required to prosecute an NCSO if an attacker is somehow moved from a production system that they have compromised to a honeypot.

Spitzner advocates for putting in place stringent control measures to ensure that a honeypot cannot be used to damage production systems; this is the second way in which honeypots are deemed inadequate for the prosecution of NCSOs. To fool the attackers, they will have to be afforded complete control over the compromised system, and the measures put in place to limit its use will have to appear to be legitimate with respect to the organization's network administration and security policies. Any attempts at restricting what the attackers can do with the compromised system, as deemed necessary in honeypot deployment, will increase the probability that attackers will discover that the compromised system is not a high value target. Furthermore, honeypot deployments do not provide activity at the HID interface level as proposed in this research.

Finally, honeypots' lack of production value makes them unsuitable for the preparation of attackers for other operations. The sophisticated attackers against whom NCSOs may be launched are likely to be knowledgeable and not easily fooled. If defenders want to prepare attackers for future operations, they will have to provide them with information that attackers deem to be of value which is by definition not found on honeypots. These three facts argue in favour of the idea that attackers will not be deceived by honeypots.

2.3. Cyber Deception

Recall that this work argues that in order to gather intelligence on attackers, we must keep them from altering their behaviour on the target system. We are therefore very interested in deceiving the attacker and we argue that this work will fit well within the corpus of literature on computer security deception. While there are many works on cyber deception in specific contexts [23] [24]and software decoys [25]; this section will review two important works from this corpus that deal with cyber deception in general. We will adopt Yuill et al.'s definition of deception *as those actions that are taken to deliberately mislead the attackers and to thereby lead them to take (or not to take) specific actions that aid computer security* [26].

Work by Yuill et al. for the US Department of Defence suggests that deception can serve an important or even indispensible role in computer security [26]. The work argues that it is important to consider computer security deception as being carried out in a large context, and therefore discusses the concept of Deception Operations. This view that deception fits within a larger context is in accordance with this work and it fits well with the NCSO that we discussed in section 1.2.2. The authors argue that there are two types of Deception Operations: those that aim to *hide the real* and those that aim to *show the false*.

The Deception Operations process is shown as a process model at Figure 4 [26], included here by permission. While a complete discussion of this process fall outside the scope of this research, a few important aspect bear notice. The process begins with the development of the Deception Operations which aims at defining the deception objective; this is to induce the target of the deception to take some action (perhaps to do nothing) and to exploit that action to one's advantage.

A deception story is presented to the target during deployment, in the target's observation arena. The target of the Deception Operation is then engaged when it receives the deception story, accepts it and takes the intended action.

Because this research describes a framework that is intended to be used in the context of a NIST deployed on a compromised system by attackers, the observation arena corresponds to the system that the attackers have compromised. This goes to demonstrate that the framework described here is a valid effort that will contribute to the field of deception for computer security.

It should be noted however that the framework proposed in this research is different from the work described by Yuill et al. in one key aspect. In contrast to this research, the work by Yuill et al. [26] presumes that the target of the Deception Operation's only view of the computer system consists of network traffic. This is not the case for those attackers being targeted by NCSOs as they have the capabilities described in section 1.3.1, including their ability to monitor and analyse HID events on the USB. This work therefore extends the concept of Deception Operations to the USB on the compromised system.

Work by Rowe of the US Naval Postgraduate School also proposes to model deception for computer security [27]. Rowe suggests that most defensive deception efforts for computer security have consisted of degrading the attackers' quality of service, thereby encouraging the attackers to move on. Rowe then improves the deception by modeling the *excuses* (reasons for denying service) given to the attackers such as suggesting there are problems with the attackers' command syntax, suggesting network delays, saying a resource is unavailable, etc. Those excuses are ranked as to their likelihood in context and one is served to the attacker. Rowe envisions such use of deception as a secondary line of defence when the security perimeter (consisting of firewalls, IDS and other security appliances) has been breached.

Those concepts have been incorporated in a honeypot prototype that is meant to attract attackers and waste their time. While such an attacker inconvenience concept is interesting, it is not directly applicable to the problem that this

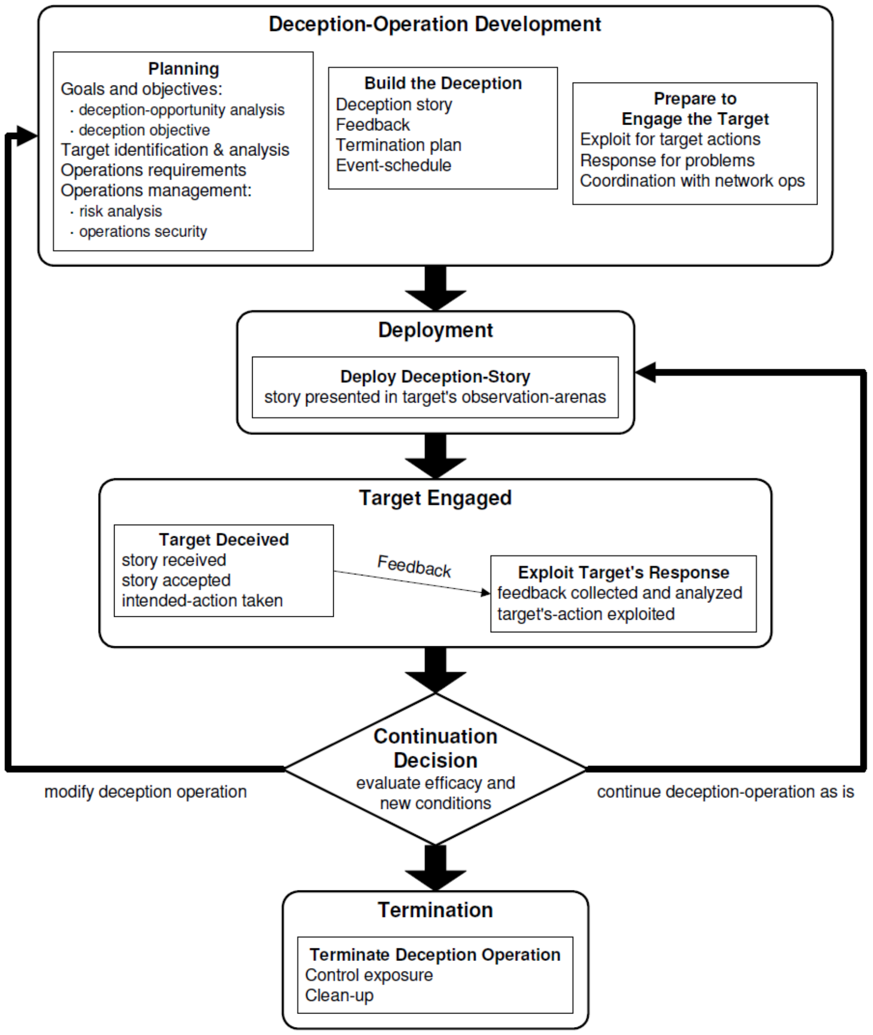

Deception-Operation Development
Planning
Goals and objectives:
· deception-opportunity analysis
· deception objective
Target identification & analysis
Operations requirements
Operations management:
· risk analysis
· operations security
Build the Deception
Deception story
Feedback
Termination plan
Event-schedule
Prepare to Engage the Target
Exploit for target actions
Response for problems
Coordination with network ops
Deployment
Deploy Deception-Story
story presented in target's observation-arenas
Target Engaged
Target Deceived
story received
story accepted
intended-action taken
Feedback
Exploit Target's Response
feedback collected and analyzed
target's-action exploited
Continuation Decision
evaluate efficacy and new conditions
modify deception operation
continue deception-operation as is
Termination
Terminate Deception Operation
Control exposure
Clean-up

Figure 4 - The Basic Deception Process [26]

research addresses because NCSO do not seek to attract attackers, rather they are carried out when defenders have already established contact. Nevertheless [27] reinforces the validity of this research effort.

Rowe also suggests the use of counter-planning deceptions to foil cyber-attacks [28]. As an example, imagine a defensive system that can detect an attack by malware and simulate infection while simultaneously rendering the malware inert. Specifically, Rowe defines work on obstructive counter-planning, which is planning to interfere with or frustrate an adversary's existing plan. To this end, he proposes MECOUNTER which is an agent-based application that reacts to attackers' actions on a compromised system in accordance with defenders' stated counter-planning deception objectives. In this work, Rowe also argues that the major flaw of honeypots is that they do not have production value; therefore, as we argued in section 2.2.4, a production system as much greater potential to fool attackers, and thereby enable intelligence collection.

In his paper [28], Rowe discusses the application of classic deception methods [29] [30] to cyber deception. Of particular interest are the seven deception methods of conventional warfare as shown in Table 2-2. As stated by Rowe, not all of these may be applicable to cyber space. Concealment and camouflage are difficult in the cyber environment because all the information present on a compromised system is visible to attackers (as per our definition of their characteristics at section 1.3.1). Similarly, realistic demonstrations and feints are hard to achieve because of the risks of not following through on a demonstrated weakness.

Table 2-2- Deception Methods of Conventional Warfare [30]

Method	Meaning
concealment	hiding friendly forces from the enemy
camouflage	hiding troops and movements from the enemy by using artificial means
demonstrations	making a move with friendly forces that implies imminent action, but which is not followed through
lies,	deliberately dishonest communications with the enemy
feints	making a move with friendly forces that implies imminent action, but following through with a different action
displays	making the enemy see what is not there
insight	deceiving the enemy by outthinking him
ruses	tricks, such as displays that use enemy equipment and procedures
false and planted information	allowing the enemy to find information that will hurt him while benefiting friendly forces

2.3.1. Contributions of Research to Cyber Deception

Rowe goes on to demonstrate how his tool, MECOUNTER focuses its efforts on lies, displays, and insights. MECOUNTER implements these three deception methods involve the analysis of the potential plans that can be carried out by attackers on a computer system and their negation through lies, displays or insights. While a detailed discussion of MECOUNTER falls outside the scope of this work, we believe that this research extends the application of classical deception methods to NCSOs.

Rowe dismisses ruses by arguing that it is not possible to surprise attackers who have compromised a computer system because of the level of control they exert over that system. In an NCSO, a compromised system is by definition an asset that is controlled by the attackers. Using a NSIT, including an HGP, on a computer system that has been compromised by attackers is an effective way of turning it into a tool to help defenders collect intelligence on the attackers (our aim as stated in section 1.6).

Section 1.3.3 introduced the concept of local HID semantics; namely the fact that attackers would be harder to fool if HID events were placed on the USB without them being semantically valid. This can still leave one to wonder however, why we go through the effort of modelling user activity at the USB HID level in order to automate it instead of simply replaying recorded HID events on the USB. This research provides the defenders this capability in order to give the synthetic user the ability to introduce false and planted information on the compromised computer system, which may well prepare the attackers for future operations outside of the NCSO.

2.4. Vitality Detection

2.4.1. Attackers' Motivation

Attackers have an interest in characterizing the systems that they compromise, especially when those are high value targets as described in our Operational Scenario at Section 1.4. The HGP described in this dissertation aims at making the compromised system continue to appear as a high-value production system [7]. Even for superficial compromises such as machines used to send spam, attackers want to know if they have stumbled on a honeypot, and will even go to the effort of creating tools to automatically identify the compromised system as a legitimately compromised host [31]. This desire to characterize compromised systems has lead attackers to organize in order to share knowledge about network defence efforts. In point of fact, a group of would-be attackers called the Phrack High Council was formed in 2002, with the stated aim of creating an underground revolution against the information security industry and the disclosure of information on security vulnerabilities, tactics and exploits because this could lead to effective defence [32].

The sentiment against honeypots is very strong in the attacker community, and emotions run high. The author noticed many fiery exchanges between would-be attackers and security researchers on the benefits of honeypot technologies and the usefulness of the information derived from it. A particularly nasty exchange between Corey and McCarty took place in 2003, were Corey demonstrates that he clearly dislikes honeypots [33]. Corey disputes the usefulness of honeypots stating that the approach is flawed because it is based upon three faulty premises:

1. Honeypot technology may be openly shared and remain effective,
2. Honeypot technology may be deployed in a hostile environment, and remain undetected, and
3. Even if detected, attackers will not target the honeypot or its operators for further exploitation.

[7] In biometrics [35] it is important to make sure that the biometric token presented is from a live human, and not copied or stolen. The term *vitality detection* is used to describe the techniques used to carry out this verification. Other terms such as *liveness detection* or *fingerprinting* are used in the literature, but wewill use *vitality detection* throughout.

Corey does not provide any evidence to support his argument for the ineffectiveness of honeypots, but McCarty addresses each of the premises [32]. McCarty retorted that honeypot technology could be effective even if openly shared because the technology is continuously evolving, because attackers vary in skill and care and because honeypots are scarce. He thinks that honeypots can be deployed and remain undetected because researchers are constantly adjusting them to make them more difficult to detect. In fact, he states that many researchers in the field consider the creation of hardware tools to make the detection of honeypots more difficult[8]; albeit not in the context of honeypots, such malicious hardware tools have been explored by the author [34]. As we will explore in Chapter 4, the creation of such hardware tools is similar to the approach implemented in the SUE, where the HID interface level mouse and keyboard activity is generated outside the compromised system. Finally, McCarty states that the risk of having the honeypot attacked might concern the honeypot operators, but not sufficiently to deter their use. This last point is of more concern to us, as the organization deploying the HGP could face reprisals or counter-intelligence operations (as discussed in section 1.5).

2.4.2. User Behaviour Modelling

As discussed in the Threat Model (section 1.3), we argue that it is likely that attackers will try to characterize the high value compromised system in terms of its user activity. When attackers have complete remote access to a compromised system, they are in a position to observe the event stream of the input devices such as a mouse and keyboard. We argue that it is reasonable for attackers to attempt vitality detection by observing this event stream. For the HGP to be effective, it must simulate the use of keyboard and mouse by a human, in a way that is consistent with real human activity by the attacker. This section will discuss how user activity can be modelled, in order to simulate the use of the mouse and keyboard. The section begins with a discussion of Biometrics in computer security; biometrics is discussed because it can be used to represent models of users, and the research relies on such user models. The two main components of the HGP are the modelling and simulation of mouse and keyboard HID events.

Modelling User Activity

Recall from Figure 2 on page 5 that the attacker will characterize user activity by comparing the user activity derived from the compromised system to models of user behaviour. These models are analogous to models used in anomaly-based intrusion detection. In order to facilitate the understanding of the articles that will

[8] As we will discuss below, many of the detection techniques used by attackers rely on differences between a honeypot's OS and a production system OS

be discussed in the following sections, we will discuss the modelling of user activity in general.

The process of creating models of user activity is illustrated in Figure 5. The model of user activity is a collection of features of the *user activity* under study. A set of user activities, representative of the task at hand, is used to derive prominent characteristics, called *Features*. The *Model of User Activity* is a set of these features.

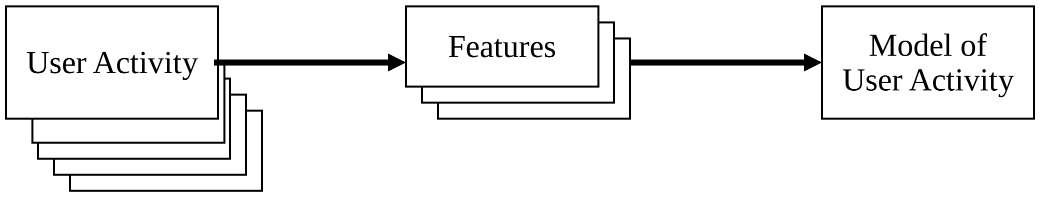


Figure 5: Generic Modelling of User Activity

The application of models of user activity is illustrated in Figure 6. In order to characterize the concrete activity of a particular user, we must ask that user to provide activity samples and analyze these samples to obtain measures for the features specified by the Model of User Activity. These measures represent a *User Activity Profile*, which is a concrete instantiation of the abstract Model of User Activity. Later, it is possible to examine a sample of *Unattributed User Activity*, obtain measures for the features specified in the Model of User Activity, and compare those to the measures found in the User Activity Profile in order to make a determination about the probability that the sample came from the user for whom we built the profile.

Biometrics is useful to the proposed research, because it represents features of users, and can thus be used to build user models. Biometrics focuses on the characteristics of the users. Those characteristics may be something unique to their physical person (*physiological biometrics*) or something they do (*behavioural biometrics*) [35]. Almost every user possesses physiological biometric features such as fingerprints, iris patterns, palm prints, and voice patterns. Similarly, features can be extracted from the different ways users interact with a computer system; such as how they type or how they use a mouse.

This research suggests that it is possible to use these features to differentiate between users to associate current session use with stored user profiles as represented at Figure 6.

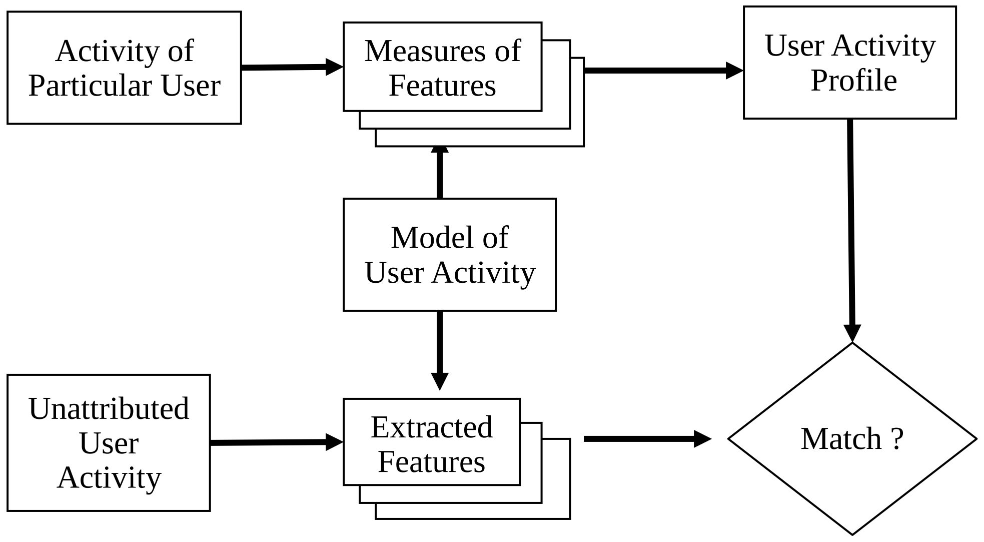


Figure 6: Attributing User Activity to Profiles

Traore and Ahmed [36] discuss the use of behavioural biometrics in the context of intrusion detection. They argue that physiological biometrics have been used to verify identity, at the start of a session in order to make access control decision, but that they have not been used for intrusion detection after initial authentication for two main reasons. First they point out that physiological biometric feature extraction requires special hardware (such as a fingerprint reader or an iris scanner), which limits the network segments on which it can be used. Second, users must provide some data sample to verify their identity; this makes physiological biometric features unsuitable for the passive monitoring that is required in intrusion detection. Behavioural biometrics features, on the other hand, do not necessarily suffer from these two shortcomings.

Modelling Mouse Dynamics

The HGP included in the NIST will need to simulate the use of a mouse by a user. It is important to review extant literature in the area of modelling mouse dynamics because attackers have access to these models. A variety of approaches have been suggested [37], but two approaches will be discussed in detail here: the research of Traore and Ahmed and the research of Pusara and Brodley.

Traore and Ahmed introduce mouse dynamics as a new biometric technology in [38] and [39] before refining the technique [36]. They hypothesize that mouse dynamics can be used to identify users of a computer system because the variations within multiple sessions of a particular user are small when compared to the variations between the sessions of different users. Mouse dynamics are defined as the characteristics of the actions received from the mouse, used as a pointing device by a specific user interacting with a graphical user interface (GUI). Those actions can be classified in one of four types: Drag-and-Drop (DD), Point-and-click (PC) general Mouse Movement (MM), or Silence where there is no movement.

Traore and Ahmed's detector captures mouse activity by creating a log record consisting of four fields: the type of action (DD, PC, MM or Silence), the distance travelled (in pixels), the duration of the action (in 1/100 of a second) and the direction of movement. The direction of movement is expressed as an integer in the range of [1-8] representing the octet in which the movement occurred; 1 for 0° - 45°, 2 for 46° - 90° and so on. The mouse activity is modelled with the seven features shown in Table 2-3. Details are given below for the seven features, which can be logically grouped in five categories: movement speed (MSD), movement direction (MDA, MDH), action type (ATA, ATH), travelled distance (TDH) and elapsed time (MTH).

Table 2-3: Mouse Activity Signature Features [39]

Factor	Ranges	Units
Movement Speed compared to Distance Travelled (MSD)	25 – 800	Pixel/Sec
Average Movement Speed per Movement Direction (MDA)	25 – 800	Pixel/Sec
Movement Direction Histogram (MDH)	1 - 8	octet
Average Movement Speed per Types of Actions (ATA)	25 – 800	Pixel/Sec
Action Type Histogram (ATH)	0 – 100	%
Travelled Distance Histogram (TDH)	0 – 100	%
Movement Elapsed Time Duration Histogram (MTH)	0 – 100	%

Because there is a large amount of data collected, the MSD is modelled by a curve over the range of distances travelled, and the particular details of this feature are stored by using 12 equidistant points along that curve. The movement speed is characterized by recording the average speed in each octant (MDA) and the proportion of all movements in each direction (MDH). A similar characterization is done for the action type features, by recording the average speed for each type of action in ATA (based on PC, DD and MM).

The ATH is a histogram that has a bin for each of the four action types, indicating the proportion of actions that fall in each bin. Similarly, the TDH gives an indication of the proportion of the distance travelled for each mouse movement; this histogram is described with nine equidistant bins from 0 pixels to the screen width[9]. Finally, the MTH illustrates the distribution of the number of actions performed by the user within different time bins during a session.

Traore and Ahmed observe that the first few bins of TDH and MTH are sufficient for classification purposes because shorter distances and times are much more likely than longer ones. This allows them some economies, because although both these factors are open-ended, the detector can classify users by using only the first two bins for TDH and the first three for MTH. It should be noted that although the system can derive Silence actions using the time elapsed in between PC, DD and MM, it does not use silences for session classification.

The detector is implemented using client/server architecture as depicted in Figure 7, included here by permission. The client portion of the system is installed on each potential target, and it is responsible for capturing mouse and keyboard data, converting them into more meaningful behavioural biometric features and passing these to the server. The server analyzes the data and computes biometric user profiles.

Like other biometric systems, the detector functions in two modes: *enrolment* where a base user profile is created, and *identification/verification* where a session profile is captured and compared against user profiles. The detector controls the duration of the mouse and keyboard data capture period. It is important to realize that the user is not alerted to the fact that data is being captured. This fact, coupled with the absence of specialized data capture hardware, enables authentication throughout the user session.

Traore and Ahmed validated their concept with a small-scale experiment to show that it is possible to detect masqueraders, defined as users pretending to be someone else. The participants (n_t = 22) installed the client software and used their PC for conducting routine activities. Their mouse and keystroke data was collected for an unreported number of sessions. The set of participants was separated into legitimate users (n_l = 10) and unauthorized users (n_u = 12). An unreported fraction of all sessions from each legitimate user was used to create a

[9] The authors used a screen 800 pixels large, thus explaining the ranges found in Table 2-3: Mouse Activity Signature Features [39].

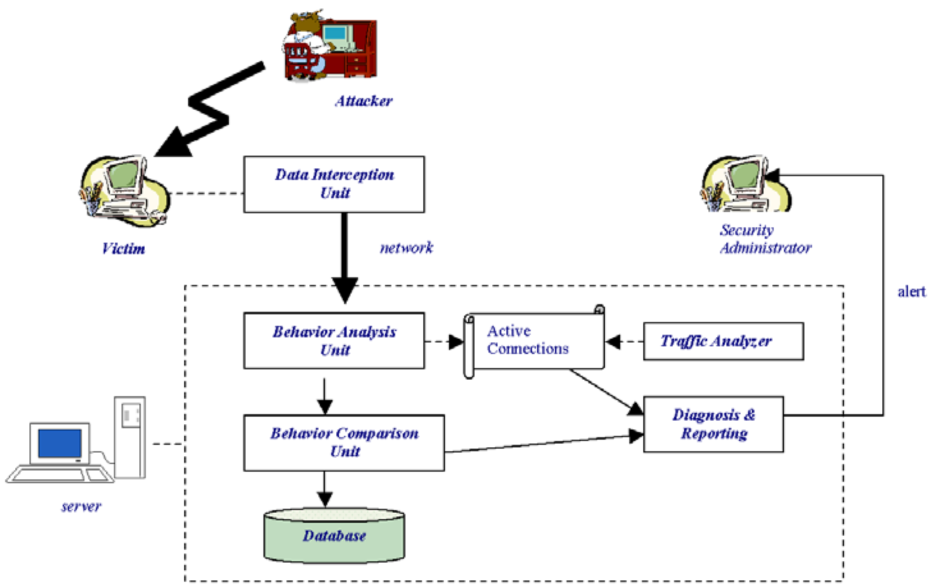


Figure 7: Detector Architecture [38]

base profile. For each legitimate user, the researchers used sessions from the other 21 users as representative masquerade attempts.

This resulted in an Impostor Pass Rate (IPR) of 0.00651. The IPR represents the proportion of impostor access attempts that are erroneously authenticated to a legitimate user of the system. The researchers also used the sessions from legitimate users that were not used to compute the base profile, and ran those through the detector. This resulted in a False Alarm Rate (FAR) of 0.01312. The FAR represents the proportion of legitimate users of the system that were not properly authenticated.

Pusara and Brodley [40] have also proposed the use of mouse movements to carry out user re-authentication. The technique uses mouse data to generate a profile for each user. The profile is built during a training phase, and later activity is compared to that profile. During the subsequent re-authentication phase, significant differences between the observed mouse activity and the profile are flagged as anomalous.

The profile is built using cursor locations (*X*; *Y* screen coordinates) sampled using a clock and also at the occurrence of mouse events which are defined using six mouse event categories represented by Figure 8, included here by permission. The Mouse Events are represented in a hierarchy with *Mouse Events* at the top level. The Mouse Events are broken down into three sub categories at the second level of the hierarchy: *Mouse Wheel*, *Clicks* and *Non-Client Area Moves*. The non-client area is defined as the area of the application window containing the menus and tool bars. The *Clicks* are further divided into *Single Clicks* and *Double Clicks* at the bottom level of the hierarchy. Each of the Mouse Events, Mouse Wheel, Clicks, Single Clicks, Double Clicks, and Non-Client Area Moves categories are used in the creation of the profile.

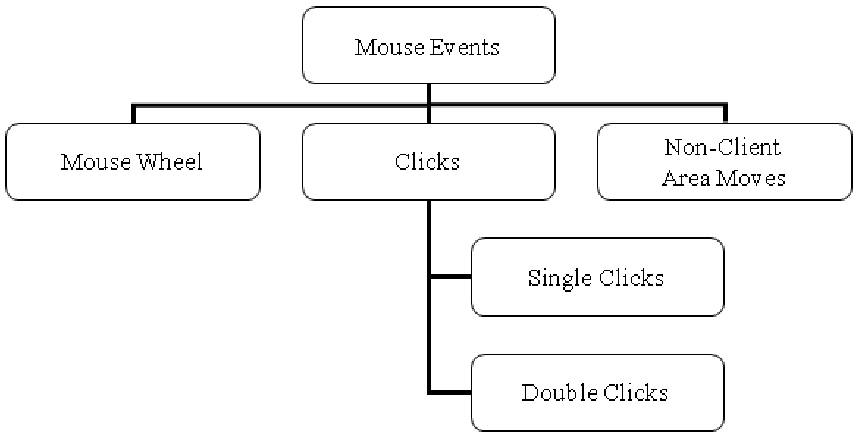


Figure 8: Mouse Event Categories [40]

The profile combines information about mouse movements (derived from the *X, Y* screen coordinates) and the six mouse event categories into numerous features. The screen coordinates are sampled every 100 ms to derive general movement data, and they are recorded every time a mouse event is detected to sample mouse event data.

Mouse movement data (either general or associated with a specific mouse event category) is derived from the cursor screen coordinates by calculating the distance, angle and speed between pairs of screen coordinates. Although the coordinates in a pair must be time-ordered, they need not necessarily be consecutive. This is accomplished by waiting for *k* data points to pass before making the distance, angle and speed calculation. This parameter *k* is called the frequency and it allows for data reduction by effectively implementing a sampling rate of $^1/_\mathrm{k}$.

The frequency value set used in Pusara and Brodley's experimentation was {1, 5, 10, 15, 20}. The amount of mouse movement data is further reduced by calculating the mean, standard deviation and third moment value of the extracted movement data (distance, angle and speed between coordinates) over a window of *N* screen coordinates. The window sizes considered during experimentations were {100, 200, 400, 600, 800, 1000}.

The technique derives the mouse event data in a similar way. For each of the six mouse event categories, it computes the distance, the angle and the speed between pairs of cursor screen coordinates (sampled at the time of the mouse events) using a specified frequency *k*. The creation of the profile for each user follows three steps:

1. *Event count*: The technique first takes a count of the number of events observed for each of the six mouse event categories. This yields six features: {Mouse Events, Mouse Wheel, Clicks, Single Clicks, Double Clicks, Non-Client Area Moves}.
2. *Mouse movement*: The technique then computes mouse movement data for each mouse event category and general mouse movement data (from the screen coordinates sampled every 100 ms). Once again, it does this by computing the mean, standard deviation and third moment value with regard to the distance, speed and angle between pairs of successive cursor screen coordinates for each mouse event category. The technique also computes the mean, standard deviation and third moment value with regard to the distance, speed and angle for pairs of screen coordinates (sampled every 100 ms) to derive general mouse movement data. This yields 63 features: {*General Mouse Movement* + 6 *Mouse Event Categories*} X {*Distance*, *Speed*, *Angle*} X {*Mean*, *Standard Deviation*, *Third Moment Value*}.
3. *Screen Location*: Finally, the technique characterizes where the mouse activity is taking place by computing screen coordinates statistics (mean, standard deviation and third moment value) for each axis (*X* and *Y*). These statistics are gathered for each mouse event category and for the general mouse movement data, yielding 42 features: {X; Y} X {*General mouse movements* + 6 *Mouse Events Categories*} X {*Mean*, *Standard Deviation*, *Third Moment Value*}.

Pusara and Brodley's technique is highly customizable because each of the $6 + 63 + 42 \ = \ 111$ features derived from the six mouse event categories and mouse movements can be calculated with a different frequency *k*. When building a profile during the training phase, frequencies can be chosen to minimize IPR or FAR during the re-authentication phase.

The technique uses binary decision tree classifiers to flag anomalous data during the re-authentication phase. A binary tree is built for each registered user of the system, where each internal node represents a test on a feature. Each of these tests can result in any two of the following three outcomes: *User Recognized*, *User Not Recognized*, or *other test node*. The leaves of the tree can only be *User Recognized* or *User Not-Recognized*.

Figure 9, included here by permission, provides an example of classification during re-authentication for a particular user (User #13). The first feature tested is the number of non-client area moves; if there are 21 or less the current session does not match to user #13. If there are more than 21 non-client area moves, the number of mouse events is tested. The algorithm used to build the decision tree falls outside the scope of the current research; it suffices to say that the tree is built in such a way as to maximize the reduction in entropy at each of its layers.

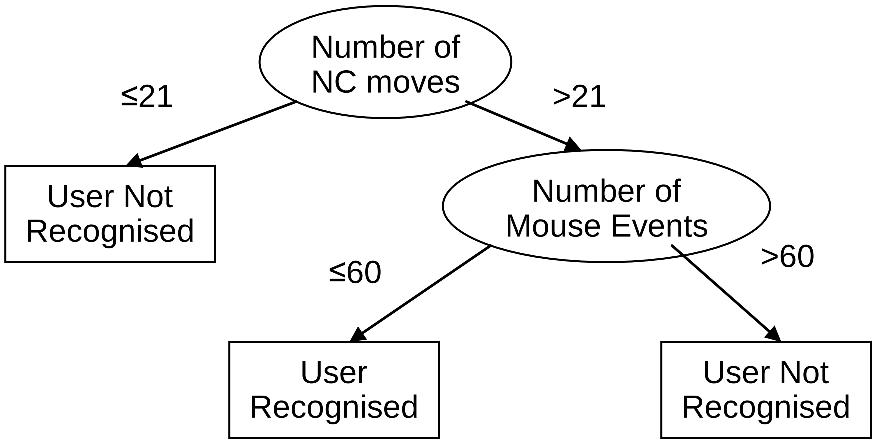


Figure 9: Decision Tree Classifier [40]

Experiments were conducted using data collected from 18 volunteer students. For each of the participants, data representing 10,000 unique cursor locations was collected, over a period of approximately two hours. The participants were using the same operating system (an unspecified version of Windows) and the same browser (an unspecified version of Internet Explorer).

Two experiments were carried out to evaluate the technique: *pair-wise discrimination* and *anomaly detection*. The pair-wise discrimination experiment was designed to evaluate the differences in behaviour between the 153 pairs of users. Although the technique was successful in discriminating between many pairs of users (101 of the pairs had error rates [IPR and/or FAR] below 0.05), it was impossible to confidently distinguish between the other 52 user pairs with eight user pairs having error rates of over 0.40.

In anomaly detection, the experiment was designed to see if a particular user *X* could be distinguished from all other users. To do this, the researchers created a supervised data set by labelling *X* data as normal and all other data as coming from intruders. Through the use of smoothing filters, the technique reports an average IPR of 0.43% and an average FAR of 0.0175. The research does not seem to have manipulated the parameters of the technique to carry out a receiver operating characteristics (ROC) space analysis. In analysing their results, Pusara and Brodley conjecture that the users having the highest FAR were those who were not making much use of the mouse.

Modelling Typing Behaviour

Following the discussion on the modelling of mouse movement, we turn our attention to typing behaviour[10]. There is a large amount of published research on the subject of typing for authentication applications [41]. As previously argued by the author however [1], that work is of little value to the current research because it does not deal with English free-text. Although we will briefly discuss general research efforts, our main focus will be on the efforts of Gunetti et al.

The Traore and Ahmed detector described earlier [38] uses keystroke dynamics as a second behavioural biometric. The biometric is analyzed by measuring the *dwell time* (length of time the key is pressed) and the *flight time* (length of time to move from one key to another) between successive keys. The relationship between the keys is recorded as either a digraph[11] or a tri-graph describing the transition between sequences of two or three keys. Traore and Ahmed offer some anecdotal evidence of the usefulness of this biometric measure, showing that it is possible to distinguish between users, but it is not clear how features are

[10] The term *Keystroke Dynamics* is used to describe the dynamics of typing text, but it usually only refers to the typing of letters. This research models the use of control characters, and will use the term *Typing Behaviour* to refer to this more general form of keyboard use modelling.

[11] We use *digraph* to represent a grouping of two letters to remain consistent with the literature [35]; the reader may think of a digraph as a *pair* of characters.

extracted from this biometric, or how these features are integrated into the detector discussed above and depicted in Figure 7.

Other common measures used to characterize typing found in the literature are the *keystroke duration*, which is the amount of time that a key is pressed (similar to Ahmed and Traore's dwell time), and the digraph latency, the interval between the depression of the first key and the depression of the second. A less common but interesting new measure called the *n-graph* duration can be found in [42]. It is defined as the amount of time elapsed between the depression of the first key and the depression of the n^{th} key of an n-graph. The *n-graph duration* is clearly a combination of the keystroke duration and digraph latencies discussed above. It is easy to see that one is able to compute the n-graph duration for any value of *n* by recording only the time at which each key is depressed.

Typing is done under many different conditions: there are many different keyboard layouts in use, the ergonomics of workstations vary greatly, and the difficulty of the text influences the typing task. Because of the limited information available to model typing biometrics, and because of the wide variability in the typing task, most research in the field uses *fixed text* which is defined as a structured target string that must be typed by the user [42]. Because typing fixed text tends to be tedious, most research uses short target strings.

As mentioned by Traore and Ahmed, there has been considerable research in the use of keystroke dynamics for authentication [38]. However, these efforts do not meet the needs of this research, because they only deal with authentication. The use of a target string is a major difference between modelling a user's typing behaviour during authentication (done at the beginning of a session) and general behaviour during a session. In authentication applications, users are generally asked to type a target string and the system analyzes the characteristics of the string being typed to compare its extracted features against stored user profiles, as depicted at Figure 6 on page 32.

The seminal work in the field of keystroke analysis was done by Gunetti and Picardi from the University of Torino in Italy (see [43] and most importantly [42]). In [42] Gunetti and Picardi offer a taxonomy of keystroke analysis based on *static*, *dynamic*, *fixed text* and *free-text*. *Static* refers to a system where a target string is typed during login, and where the system analyzes the target string after it has been typed to compare its extracted features to those stored in user profiles. *Dynamic* analysis implies that the keystrokes are being monitored during the session, after the login. However, most dynamic experiments reported in the literature indicate that the user is still asked to type a target string (or one of a small number of predetermined strings) during interruptions in the user’s session.

In dynamic authentication, the analysis may be done before the user has finished entering the target string; this is not the case for static analysis done at the beginning of a session.

In both static and dynamic analysis, the user is asked to enter some form of fixed text. When the text to be analyzed is not constrained in any way (as is the case after login during a session), the term free-text analysis is used. The HGP described in Chapter 4 mimics free-text; it must therefore be modeled.

The typing profiles found in most authentication applications (see [44] [45] [46] [47]) contain features based on digraph latency and/or keystroke duration. These features are usually represented as statistics such as means and standard deviations. When users are challenged for authentication, they provide a typing sample from which sample statistics are extracted. Those sample statistics are compared to the profile statistics, and a decision is usually made based on the result of a statistical test. An alternate technique [48] consists of representing the profile as a vector, where each element is the latency of an n-graph shared by the profile and the sample under test. Such a system can determine the probability that a sample belongs to a profile by computing a Euclidean distance between the sample and the profile. These applications require large samples to be able to make statistically valid authentication determinations.

Gunetti and Picardi represent samples as sets of pairs {*n-graph*s, *n-graph duration*}. If the same n-graph is found more than once in a typing sample, the corresponding pair uses the mean duration. As we will discuss later, this allows the approach to handle small sample sizes. Consider this example taken from [42], where two samples **E1** and **E2** are constructed from the typing of the word `authentication` and the word `theoretical`. The samples would contain the n-graphs obtained from the following letters (along with their duration):

E1: {`a, u, t, h, e, n, t, i, c, a, t, i, o, n`}
E2: {`t, h, e, o, r, e, t, i, c, a, l`}

Testing Samples through Distance Measures

Recall from Figure 6 that the features of the user activity model measured from the unattributed user activity must be compared to stored user profiles to make a determination about their provenance. Gunetti and Pacardi test the unattributed typing activity against user profiles by calculating two measures of the *distance*[12].

[12] This *distance* is used to see how closely the observed activity matches the stored user profiles; it does not correspond to any physical aspect of the typing behaviour.

In order to calculate the distance between two samples, the technique extracts a subset of {*n-graphs*, *n-graph duration*} pairs; only those pairs that are shared between the samples are used to compute the distance. The technique uses two different distance measures to compare the unattributed activity and the user profiles: an *absolute* and a relative *measure*. These distance measures are the features used in Gunetti and Pacardi's user profiles.

The relative measure (*R*) gives an indication of the relative speed at which users type n-graphs. The system extracts the n-graphs shared by two samples, and orders them according to their duration. *R* is calculated based on the relative ordering of the shared n-graphs in the two samples being compared. Although the details of the calculation of the R value will not be explained in this document, R is represented as a real value between 0 and 1. The reader should note that R measures completely ignore typing speed. The premise to their use is that if a user's typing ability is impaired by environmental or physiological factors, all n-graphs will be similarly affected, and their relative ordering will remain consistent.

For two samples, a single *R* distance measure is computed for each n-graph type, and each *R* distance measure gives an indication of the relative ordering of all n-graphs shared by the two samples; R_2 for all shared digraphs, R_3 for all shared tri-graphs, and so on. It is common for two samples to share more than one n-graph type. When this is the case, it is possible to calculate a cumulative distance between the samples by combining the individual *R* measures.

The absolute measure (*A*) only considers the speed at which n-graphs are typed. The system examines each shared n-graph and determines if its mean speed in one sample is similar to the mean speed of the same n-graph in another sample. The test of similarity was determined through experimentation. The *A* value does not give a direct indication of the speed at which a particular n-graph is typed. Rather, it is calculated by looking at the ratio of similar n-graphs to the total number of shared n-graphs of a given type. Both single *A* measures (such as A_2, A_3, etc) and cumulative *A* measures (such as $A_{2,3}$) can be computed.

The reader will note that the technique only uses the mean duration, and not any other n-graph statistics, such as the standard deviation or third moment value. While this implies that the system does not make use of all the information contained in typing samples, it allows Gunetti and Picardi to use n-graphs that occur only once in a sample.

An exhaustive discussion of relative and absolute distance calculations, including a worked example, can be found at [1]. While those details will not be represented here, the use of such measure can be used by attackers to characterize a compromised system. The implication will be discussed during the descriptions of the *Typing Accuracy Model* in section 3.5.1 and of the *HID Timing Model* in section 3.10.1.

We have seen that systems that only perform authentication do not provide sufficient insight into keystroke analysis for the purposes of this research's HPG. Gunetti and Picardi suggest a hierarchy of keystroke analysis tasks, as depicted in Table 2-4. The table shows the name of the task, the possible provenance of the sample (*X*), where *U* and *V* are known users of the system, and *O* represents an outsider. The table also lists the claim made by the user providing *X* and the decision that the system must render.

Table 2-4: Keystroke Analysis Task Hierarchy [42]

Task Name	Provenance of *X*	Claim made by *X*	System Decision
Classification	*X* comes from one of the known users of the system	Nil	Which *U* provided *X*
Authentication	*X* comes from *U*, or *X* comes from *V*, or *X* comes from *O*	*X* claims to belong to user *U*	Truthfulness of the claim
Identification	*X* comes from *U*, or *X* comes from *O*	Nil	*X* comes from *U*, or *X* comes from *O*

The hierarchy presented in Table 2-4 is based on an increasing level of difficulty, with *Classification* being easiest. Recall that this research is aimed at producing a framework for the generation of HID events (see section 1.6) in a way that is consistent with that of a real user, from the perspective of attackers that have certain capabilities (see section 1.3). We argue that it is necessary for the generated activity to be consistent with what would be generated by a *particular* user, which means that the HGP described in this research would pass the *Identification* test.

We take pause to review our analysis of User Behaviour Modeling. We have clearly established that modelling user activity is a premise of the research. The research efforts presented in this chapter are useful in gaining insight into some of the requirements of the HGP. As we recall from the threat model in Section 1.3, attackers use models of user behaviour to characterize user activity on the compromised system. We presume attackers to have access to extant literature;

this implies that the proposed research must take that literature into consideration.

Behavioural biometrics can help define user models, and we have seen how the features contained in these models can be used to build user profiles. Within the scope of the proposed research, we have examined the modelling of mouse and typing dynamics.

Major research efforts have been examined in the context of modelling mouse dynamics [36] [38] [39]. None of these is directly applicable to the definition of models of user activity necessary to the HGP. The work by Gunetti & al. is clearly applicable to the research, because the proposed HGP must simulate seemingly free text.

All of these approaches have a shortcoming in that they model mouse activity and typing activity for individual profiles, but they do not characterize it across users. To that end, our research goes beyond the modelling of mouse and typing dynamics, and requires more general models of mouse use and typing behaviour.

2.5. Chapter summary and conclusion:

The HGP for which this research describes an HGP framework would be useful in the conduct of NCSOs. This chapter has reviewed current research literature as it applies to the research.

The chapter defined honeypots as a security resource whose value lies in being probed, attacked or compromised (section 2.2.1). While honeypots may be useful for capturing broadly targeted malware, they cannot help with the collection of intelligence on a sophisticated attacker as described in the Operational Scenario (section 1.3). A honeypot does not have any production value; the HGP is then clearly not a honeypot because it is meant to be used on a compromised production system after the compromise is discovered.

By its very nature, a NCSO is a Deception Operation because it seeks to obtain information on attackers without their knowledge. The review of literature on cyber deception (section 2.3) demonstrates that deception is an active avenue of research, which goes to reinforce the validity of the research topic.

The research into vitality detection and the modeling of user behaviour discussed in section 2.4 is readily available to attackers. It stands to reason therefore, that attackers may use it to characterise a compromised system which they deem to be of high value. The work discussed and reviewed in this chapter demonstrates that vitality detection by attackers is a worthwhile problem. The stated research aim stated in section 1.6, namely *to develop a conceptual framework for the automatic generation of HID events in a manner that is consistent with a human inputting text into a computer system*, represents a worthwhile contribution to the field of computer security.

CHAPTER 3 : HID EVENT GENERATION PROCESS

3.1. Chapter introduction

This research presents a framework which allows for the automatic generation of HID events on a compromised system. The generated HID events are those associated with the use of a mouse and keyboard by a human user using a text entry application, such as an email client or simple text editor. The framework, which we term the *HGP*, is presented in detail in this chapter.

While the act of recording HID events and replaying them on the USB would be sufficient to make them consistent with those generated by a human user, that this is not sufficient for the conduct of NSCOs, because attackers have the ability to examine the semantics of the text being entered as detailed in section 1.3.3. Simply replaying HID events could well result in an HID event stream that is not meaningful in the context of the computer system compromised by the attackers. The HGP described here therefore allows for the generation of an HID event stream that is consistent with a human user entering arbitrary text in the compromised computer system.

Because attackers are assumed to have complete control of the compromised system, they can observe what takes place on it. It is not possible therefore for the generation of HID events to take place on the compromised computer system, as the generation activities would be visible to attackers. The automatic HID generation must therefore take place off the compromised system to be subsequently placed on its USB.

Such an HID event stream can be useful in the context of an NCSO, and it can also help prepare the attackers for further operations, including *Deception Operations*. The chosen arbitrary text for which HID events are automatically generated by the HGP can represent *disinformation* that can be used in support of many of the deception methods discussed at Table 2-2. Entering arbitrary text which the attacker can see is, by its very definition, the provision of *false and planted information*. In the greater context of a campaign against an organized adversary, the compromised system can be used to support wider *Deception Operations* by helping with *lies*, *displays* and *feints*. While it would be possible to simply plant the disinformation material on the compromised system, for example by saving a file in the system’s documents repository, attackers who have complete control of the compromised system may notice the appearance of new documents and become suspicious; the ability to make it appear as though the disinformation is being input into the compromise system, as provided by the HGP, can be of use to network defenders.

The main purpose of this chapter is to present the HID Event Generation Process; we will use HGP or *process* unless the context makes this ambiguous. The chapter begins with a discussion of models of user behaviour on which the process depends. This is followed by an overview of the process itself, leading to a detailed discussion of the seven stages that implement it. We then comment on the process from the point of view of the attackers before summarising the chapter.

3.2. User Personality Model

The major contribution of this research is a framework for the automatic generation of HID events on the USB. Recall from section 1.6 that we aim for that generation to be consistent with that of a human using the compromised system. Section 2.4 has argued that attackers will go to great lengths to characterize the computer systems that they have compromised, especially if they deem those to be of high-value, such as systems for which defenders would launch an NCSO. This begs the questions: how will attackers decide if the activity they see on the USB is consistent with that generated by a human user? Although the open literature does not detail any such models, it is reasonable to presume that attackers will build models of user activity, and the process is premised on such models.

We use the term *User Personality Model* to describe the set of models of user activity that attackers could use to characterize user HID activity on a system that they have compromised. The User Personality Model is composed of five components, as depicted at Figure 10 - The User Personality Model.

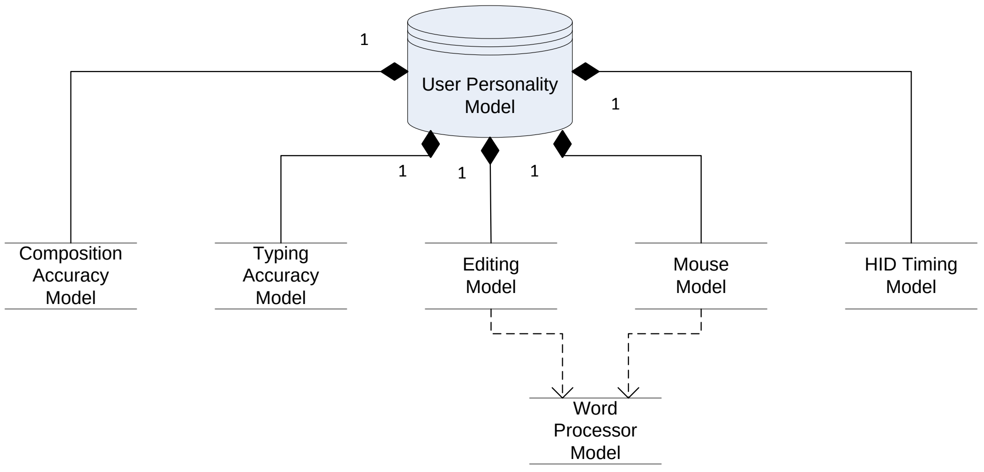


Figure 10 - The User Personality Model

The *Composition Accuracy Model* is a probabilistic model that describes the synthetic user's likelihood of making *Composition Errors* when composing text. The composition errors modelled are the misplacement of a syntactic element (such as a `Word`[13], `Number`, `Sentence`, `Paragraph`, etc.) and the use of the wrong syntactic element. The Composition Accuracy Model lists the possible errors, along with their probability of occurrence, and it is discussed in more detail in section 3.5.1.

The *Typing Accuracy Model* is a probabilistic model of the accuracy of the synthetic user using a keyboard. For each digraph or tri-graph, the *Typing Accuracy Model* lists the discrete probability distribution that the synthetic user will type the intended digraph/tri-graph or an erroneous one. The Typing Accuracy Model is discussed in more detail in section 3.5.1.

The *Editing Model* is a probabilistic model that describes the synthetic user's choices in terms of which *Editing Actions* are performed. There are many different ways to accomplish editing tasks (such as selecting text and using an application's interface). The Editing Model describes the synthetic user's preferences regarding editing actions and expresses those preferences as a discrete probability distribution of keyboard and mouse actions. The Editing Model is described in more detail during our discussion of Stage 3 in section 3.6.1.

The *Mouse Model* is a probabilistic model of the use of the mouse by the synthetic user in terms of movement speed and button use, along with potential mouse errors. It divides mouse errors in one of two categories: those associated with *Mouse Movements*, such as missing the aimed location on the screen (`Missed_Loc`) and *Mouse Button Use* such as the accidental pressing of a button (`Accident_But`), and the use of the wrong mouse button (`Wrong_But`). The Mouse Model will be discussed in more detail in our discussion of Stage 5 in section 3.8.

The *HID Timing Model* is a probabilistic model describing the distribution of delays between various HID events. The model described the elapsed time between the Keyboard Output Reports (`KORs`) and Mouse Reports (`MORs`) being placed on the USB. The HID Timing Model will be discussed in more detail in our discussion of Stage 7 in section 3.10.

---

[13] By convention, we will use fixed-width font to represent elements found in a *lexicon*.

Figure 10 shows that the *Editing Model* and Mouse Model depend on the *Word Processor Model* (discussed at section 3.7.1). This stands to reason as the choice of editing action and the location of various word processor application controls (and therefore the targets of *Mouse Movements*) are dependent on the word processor used to compose text. While these dependencies are mentioned here, they will be explored in more detail in section 3.7.1.

There has been significant research in the use of the HID from the field of human-computer interaction (HCI). Specifically, the *Goals, Operators, Methods and Selection rules* (GOMS) family of models [49] is known to be one of the most widely known theoretical concept in HCI and the models of pointing tasks based on Fitts Law are useful for analysing pointing tasks [50]. The User Personality Model that we propose here is not meant to replace seminal HCI works; rather it should be taken to be a reasonable approximation of the models that may be used by an attacker in characterizing user activity. We use the User Personality to argue that the HGP described in this research can integrate a wide variety of such models of user behaviour.

Recall from our Operational Scenario (section 1.4) that the application that will implement this HGP is meant to be used in the context of a NIST where it will represent a synthetic user who will assume the role of the regular user of the compromised system at the HID. In order to ensure that the synthetic user can approximate the HID behaviour of the regular user, the *User Personality Models* must be parameterized to exhibit different HID behaviour.

These models are discussed in more detail during the discussion of the process stage where they are first used; the complete definition and population of these models falls outside the scope of the research. This represents an important avenue of future work as discussed in section 5.6.

3.3. HID Event Generation Process Overview

The HGP is depicted at Figure 11. It analyses a `TargetDocument` in order to derive a sequence of HID events (`MORs` and `KORs`) simulating a human user composing the `TargetDocument`. The HGP works with the representation of the `Document` being produced on a word processor. The `Document` representation is dynamic, beginning with an empty `Document` at the start of Stage 1 to eventually contain the sequence of HID events that will render the composition of the `TargetDocument` after Stage 7.

The process is based on a pipes and filters architecture where each successive stage accepts an input from a previous stage, transforms this input and forwards the resulting output to a subsequent stage. By its very nature, a pipes and filter architecture is modular; this allows the process to be refined and extended. Stages can be replaced, modified or added as long as the stages produce outputs that will be acceptable as input to follow-on stages. The first input to the process is a `TargetDocument` which is the `Document` that is to be rendered using HID events. The final output is a sequence of pairs (HID event, inter-event delay) which, when placed on the USB, make it appear that a human user is composing the `TargetDocument`.

Each stage produces a *Document Production Model* (DPM) which describes the sequence of transformation needed to produce the output from the input. It is important to keep in mind that the DPMs do not represent a `Document`, but rather the steps (or *productions*) necessary to generate a sequence of HID events which, when placed on the USB of the compromised system, would be consistent with a human user composing the `TargetDocument` using HID.

Not many users are able to compose text, even a simple `Document` such as a short email, without making mistakes. The reader will therefore notice that Stages 2 and 5 of the HGP are devoted to the introduction of errors in the DPM, such that the synthetic user will appear to make mistakes, thereby respecting the local semantics considerations discussed in section 1.3.3.

Each stage of the HGP uses a *lexicon*, which defines the syntax of the transformation language understood by the stage. Each lexicon is an extension of the previous lexicon, which means that every stage understands the syntax of all pervious stages. This is important because we use unrestricted ambiguous context free grammars (as represented by the lexicons) to manipulate English free-text, as will be discussed further in 4.2.

Two further models are represented on Figure 11. The *Document Model* (`DocMod`) is the evolving representation of the `TargetDocument` as it is being composed by the synthetic user on the compromised system; one can think of this representation as corresponding to the text composition area of a word processor application. The `DocMod` will therefore contain the text as it is entered, including the errors and corrections of these errors necessary to ultimately render the final version of the `TargetDocument` on the compromised system. The synthetic user will be composing the `TargetDocument` using an application on the compromised system, as represented by the *Word Processor Model* (`WordProcMod`). The `WordProcMod` formalises the word processing application functionality, such as the text entry, manipulation and navigation, so that it may be used by the HGP.

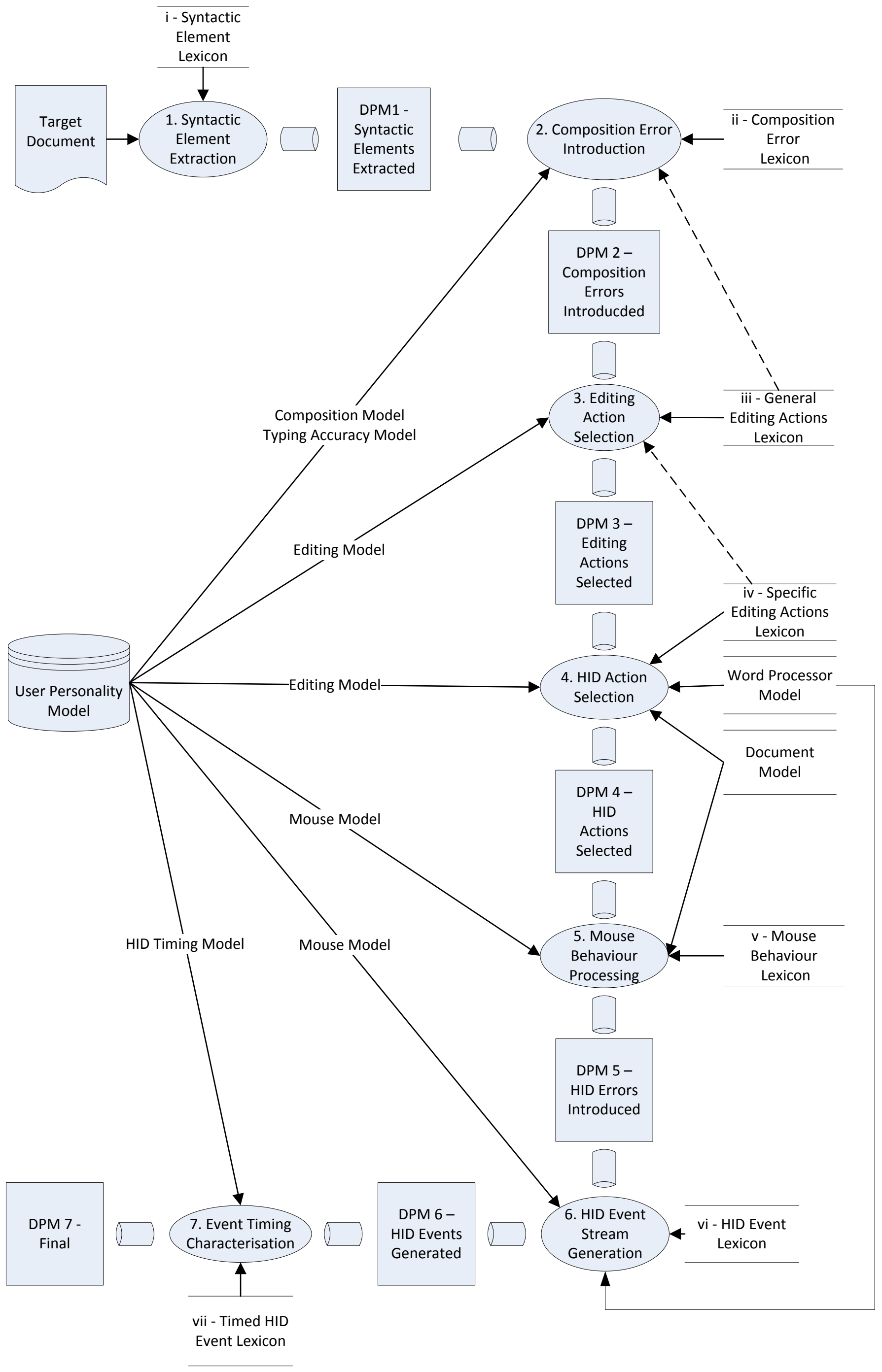

i - Syntactic Element Lexicon
Target Document
1. Syntactic Element Extraction
DPM1 - Syntactic Elements Extracted
2. Composition Error Introduction
ii - Composition Error Lexicon
DPM 2 – Composition Errors Introducded
Composition Model
Typing Accuracy Model
3. Editing Action Selection
iii - General Editing Actions Lexicon
Editing Model
DPM 3 – Editing Actions Selected
iv - Specific Editing Actions Lexicon
User Personality Model
Editing Model
4. HID Action Selection
Word Processor Model
Document Model
Mouse Model
DPM 4 – HID Actions Selected
HID Timing Model
Mouse Model
5. Mouse Behaviour Processing
v - Mouse Behaviour Lexicon
DPM 5 – HID Errors Introduced
DPM 7 - Final
7. Event Timing Characterisation
DPM 6 – HID Events Generated
6. HID Event Stream Generation
vi - HID Event Lexicon
vii - Timed HID Event Lexicon

Figure 11 - The HID Event Generation Process

3.4. Stage 1 – Syntactic Element Extraction

The HGP simulates the generation of a `TargetDocument`, which is a specific instance of a `Document`. A `Document` can be described by a sequence of *Syntactic Elements* and the structure between them. The first stage of the HGP captures these syntactic elements and the structure between them in a format that can be manipulated by follow-on stages.

3.4.1. Input: TargetDocument

The HGP begins with the processing of a `TargetDocument` which represents the final version of the arbitrary text that is to be composed on the compromised system by the synthetic user. In the context of the composition of an email, the final version would be what is sent by the user.

3.4.2. Dependency: i – Syntactic Elements Lexicon

The *Syntactic Elements Lexicon* (`i-SynElmtLex`[14]) describes those syntactic elements that are recognized by the HGP. While there are no restrictions on the syntactic elements of English free-text, the `TargetDocument` must be specifically structured to be recognized by Stage 1 of the HGP:

- The `TargetDocument` cannot contain any numbered or bulleted lists such as this one,
- The `TargetDocument` can only contain the following syntactic elements: `Character`, `Digit`, `Document`, `Letter`, `New_Line`, `Number`, `Paragraph`, `Punctuation`, `Signature`, `Sentence`, `SentenceElement`, `Separator`, `Word`.
- The last line of the `TargetDocument` must contain a tag indicating the end of text, etc.

The complete definition of each of the syntactic elements listed above can be found at Appendix A; representative examples of the elements of `i-SynElmtLex` can be found in Table 3-1.

[14] All lexicons are numbered using Roman numerals; all stages and their corresponding DPM are numbered with Arabic numerals.

Table 3-1 - Syntactic Elements Lexical Examples

Syntactic Element	Definition
`Word`	1 or more `Letters`
`Sentence`	1 or more (`Words` or `Numbers`) and `Punctuation`
`Paragraph`	1 or more `Sentence` and `New_Line`
`Document`	1 or more `Paragraphs` and *optional* `Signature`

3.4.3. Transformation 1 – Identification of Syntactic Elements

The HGP parses the `TargetDocument` to identify syntactic elements. The aim of this transformation is to parse the `TargetDocument` in order to recognize all the syntactic elements contained therein and to store them as a tree that can be further manipulated by follow-on stages. The syntactic elements are extracted in the following manner:

1. Beginning with a `Document` containing a single `Paragraph` with an empty `Sentence`, the HGP builds a tree[15] representing the structure of the `TargetDocument.`
2. Each node of the tree will represent a syntactic element and the tree hierarchy captures the structure between nodes. Each node will identify the type of syntactic element (such as `Word`, `Punctuation`, etc.) and its specific instance (such as `"The"`, `"."`, etc.).
3. Each `Paragraph`, `Sentence`, `Word` and `Number` will be annotated with a unique syntactic element identifier (`SE_ID`). This `SE_ID` will be used to identify the specific syntactic element in further transformations.

3.4.4. Output: DPM1 – Syntactic Elements Extracted

The output of the first stage of the process will be a tree with `Document` at its root, `Paragraphs` and `Signature` at the next level and `Word`, `Number` and `Punctuation` as leaves.

[15] The DPMs manipulated by every stage of the HGP will be represented as trees.

3.4.5. Worked Example

Recall our discussion of local event stream semantics in section 1.3.3. We will use the example presented at Figure 3 to demonstrate the stages of the HGP.

In this example, the `TargetDocument`, which would be sent by the synthetic user as an email, is:

```
Here is a great sentence:
The quick brown fox jumps over the lazy dog.
Sly
```

Stage 1 begins with an empty `Document` like this one:

```
PARAGRAPH 1 [
SENTENCE 1 [ ] ]
```

This stage's transformation will parse the `TargetDocument` and insert nodes in the DPM as syntactic elements are recognised until in the following DPM is obtained:

```
PARAGRAPH 1 [
SENTENCE 1 [WORD 1 [Here] WORD 2 [is] WORD 3 [a] WORD 4 [great] WORD 5
[sentence]:]]
PARAGRAPH 2 [
SENTENCE 2 [WORD 6 [The] WORD 7 [quick] WORD 8 [brown] WORD 9 [fox] WORD
10 [jumps] WORD 11 [over] WORD 12 [the] WORD 13 [lazy] WORD 14 [dog].] ]
SIGNATURE [WORD 23 [Sly]]
```

3.5. Stage 2 – Composition Error Introduction

A human user does not create a `Document` perfectly on its first attempt; the composition of the `Document` is usually fraught with errors and corrections. Stage 2 of the process accepts the tree of extracted syntactic elements from the previous stage, and transforms it to include errors and their corrections. To generate believable user activity, the HGP must simulate this imperfect behaviour, as modeled in the *User Personality Model*. The second stage of the process annotates the syntactic elements to allow for the introduction of errors.

The HGP considers two types of errors: *Mouse Errors* representing mistakes in the use of the use of the mouse (as discussed in detail in section 3.8) and *Composition Errors* representing mistakes in the typing, choice or positioning of syntactic elements. Some errors and their associated corrections have strong dependencies on other syntactic elements and the HGP must consider these dependencies using a `DocMod`. In an effort to minimize the correlation between HGP stages, Stage 2 does not consider errors associated with the structure of the `Document` or the word processor application used. Such errors are described in the DocMod and WordProcMod respectively and processed in Stage 4 at Section 3.7. It should be noted that the HGP does not consider recursive errors; in other words the synthetic will not make new errors while correcting previous ones.

3.5.1. Dependencies

ii – Composition Error Lexicon

The *Composition Error Lexicon* (ii-CompErrorLex) describes the *Composition Errors* that are considered by the HGP. The ii-CompErrorLex augments the previously defined i-SynElmtLex by introducing new nodes and redefining others. The main elements introduced in ii-CompErrorLex are *Composition Errors* (CompError) and Correction.

When describing errors, we distinguish between the *Target Syntactic Element*, which is the error-free representation of the syntactic element found in the TargetDocument and the *Erroneous Syntactic Element*, which represents the application of the error to the target syntactic element. For example, in a mistyping error where the synthetic user mistakenly types “Teh” when he intends to type “The”, the *Target Syntactic Element* is “The” and the *Erroneous Syntactic Element* is “Teh”.

CompErrors can be applied to Word, Number, Sentence, or Paragraph and they include:

- *Misplaced Syntactic Elements* (Misplaced_SE) where a syntactic element is entered either too early (to the left of the target syntactic element) or too late (to the right of the target syntactic element) during text composition,
- *Mistyped Syntactic Elements* (Mistyped_SE) where the target syntactic element is typed incorrectly, and
- *Wrong Syntactic Elements* (Wrong_SE) where the erroneous syntactic element is typed instead of the target syntactic element.

Each error captures five elements:

1. A unique *Error ID* which is used to associate an error node and a correction node in the DPM tree.
2. The *Error Details* which specifies how the target syntactic element is misrepresented.

3. The *Rectifying Actions* detailing the sequence of editing actions required in order to transform the erroneous syntactic element into the target syntactic element.
4. The *Correction Point*, specified in terms of the target syntactic element's context. It is represented as the DPM node after which the rectifying actions will be carried out. It may be any of the following:
   - `EndOfWord`
   - `EndOfNumber`
   - `EndOfSentence`
   - `EndOfParagraph`
   - `EndOfDocument`
5. A `ProcessedTag` to indicate the actions taken to deal with the error. These tags are necessary to minimize dependencies between nodes of the DPM; storing this information in the error node avoids unnecessary searches of the DPM tree in subsequent processing. Possible `ProcessedTags` are:
   - `CorrectionPointInserted`
   - `StubInserted`

The complete description of the `ii-CompErrorLex` can be found at Appendix B but sample lexical elements can be found at Table 3-2.

Table 3-2 - Composition Error Lexical Examples

`CompError`	**Error ID**	**Error Details**	**Rectifying Actions**	**Correction Point**	**Processed Tag**

Misplaced Syntactic Element	Misplaced_SE _Unique#	`Position {Direction, Magnitude}`	One of:
			`Move_SE` or `Delete_SE` and `Type_SE`
				One of:
				`EndOfWord`
				`EndOfNumber`
				`EndOfSentence`
				`EndOfParagraph`
				`EndOfDocument`
					`Stub Inserted`

Mistyped Syntactic Element	Mistyped_SE_ Unique#	Erroneous syntactic element as one of:
		`Word`
		`Number`
			`Replace_SE`	One of:
				`EndOfWord`
				`EndOfNumber`
				`EndOfSentence`
				`EndOfParagraph`
				`EndOfDocument`
					`Correction Point Inserted`

CompError	Error ID	Error Details	Rectifying Actions	Correction Point	Processed Tag
		Sentence Paragraph			
Wrong_SE	Wrong_SE_Unique#	Erroneous syntactic element as one of: Word Number Sentence Paragraph	Replace_SE	One of: EndOfWord EndOfNumber EndOfSentence EndOfParagraph EndOfDocument	Correction Point Inserted

Recall that the HGP must generate HID actions necessary to produce a TargetDocument; Corrections must therefore be applied to generate the HID events necessary to fix the errors that are introduced. Corrections are applied at a particular point in the DPM (the CorrectionPoint) which represents the moment at which the synthetic user recognizes the error and applies the specific rectifying actions to correct it. The CorrectionPoint must follow the erroneous syntactic element in the DPM, because it is not possible to notice or rectify an error before it has been made.

Composition Accuracy Model

The *Composition Accuracy Model* is a probabilistic model that describes the synthetic user's likelihood of making CompErrors when composing text. Recall that the CompErrors modelled are the erroneous typing of a syntactic element (discussed in the following *User Personality Model* component), the misplacement of a syntactic element and the use of the wrong syntactic element. While we believe that the modeling of composition errors is a likely avenue of characterization by attackers, we have not found descriptions of such models in the research literature. We have decided therefore, to describe a representative *Composition Accuracy Model* and use it in the development of the HGP.

The *Composition Accuracy Model* that we envision will detail the Misplaced_SE and Wrong_SE errors, and introduce Correction Points as detailed in the paragraphs below. It must be possible to parameterise the Composition *Accuracy Model* in order to have the synthetic user approximate the HID behaviour of the regular user of the compromised system, and this parameterisation is possible through the use of different probability density functions.

The *Composition Accuracy Model* will contain a discrete probability distribution on the number of `Misplaced_SE` errors by type of syntactic element (`Word`, `Number`, `Sentence` or `Paragraph`) that can be made by the synthetic user. For each type of syntactic element, the *Composition Accuracy Model* will also contain a discrete probability distribution function (PDF) of the `Direction` of the error (either `Left` or `Right`) along with its `Amplitude` expressed in syntactical element units; `Words/Numbers` may be misplaced by a certain number of `Words/Numbers`, `Sentences` may be misplaced by `Sentences` and `Paragraphs` by `Paragraphs`.

Table 3-3 gives an example of a representative *Composition Accuracy Model* representation of `Misplaced_SE` error probabilities where each row represents the target syntactic element type on which the error is modelled, and each column represents the probability of occurrence (in the first half of the table) and the `Direction/Amplitude` of the error (in the second half of the table). Note that the `Direction/Amplitude` PDF is not considered in those cases where there is no actual occurrence of a `Misplaced_SE` error.

Table 3-3 - Composition Accuracy Model - Representative `Misplaced_SE` PDF

Misplaced_SE	Number of Occurrences			`Direction/Amplitude`			
Erroneous Syntactic Element Type	**0**	**1**	**2**[16]	**Left/1**	**Right/1**	**Left/2**	**...**
Word/Number	0.40[17]	0.32	0.11	0.29	0.45	0.02	...
Sentence	0.80	0.06	0.01	0.15	0.40	0.06	...
Paragraph	0.92	0.01	0.00	0.20	0.45	0.02	...

To model `Wrong_SE` errors, the *Composition Accuracy Model* will use a discrete PDF and a dictionary containing lists of synonyms and antonyms for each syntactic element of interest as shown in Table 3-4. Each row of that table shows the target

[16] Note that in all tables containing PDFs, we do not show probabilities for every value of the random variable; all values would be required in fully populated models.
[17] While we have defined the components of the *User Personality Model*, we have not conducted experiments to populate them with data. All probabilities shown in the tables that follow are meant to be representative examples, and not indicative of actual empirical values.

syntactic element and each column shows the *Candidate Syntactic Element* that may be substituted for the target along with their respective probability.

Table 3-4 - Composition Accuracy Model - Representative `Wrong_SE` PDF

	Target / Probability	Candidate 1 / Probability	Candidate 2 / Probability	Candidate 3 / Probability	...
great	great	good	healthy	bad	...
	0.86	0.08	0.03	0.01	...
Supervisor	Supervisor	Manager	Director	Employee	...
	0.82	0.08	0.03	0.01	...
Secretary	Secretary	Executive Assistant	Administrator	Clerk	...
	0.75	0.12	0.05	0.02	...
...	...	...	...	...	...

Because the HGP will ultimately produce the HID events that will render the `TargetDocument`, errors must be rectified. To allow the HGP to generate the HID events necessary for the rectification of `CompErrors`, the *Composition Accuracy Model* must provide a discrete PDF of rectifying actions. This is illustrated in Table 3-2 where the `Misplaced_SE` error may be rectified by either moving the erroneous syntactic element or deleting the erroneous syntactic element and typing the target syntactic element.

Upon discovering that an error has been made, a human user composing a `Document` will decide when to fix the error. It goes without saying that an error cannot be fixed before it is made, but it may not be fixed immediately after having been made. For example, while the realization that '`teh`' was erroneously typed when '`the`' was intended may be fixed immediately after the user makes the error; a more subtle error such as typing '`that`' instead of '`which`' may not be realized until final review of the `Document`. We use the term *Correction Point* to describe the point at which the synthetic user discovers the error and carries out the rectifying actions necessary to fix it.

The final element of the *Composition Accuracy Model* is the selection of these `CorrectionPoints`. We propose to model this selection as a discrete PDF of the possible correction points as shown in Table 3-5. The reader should note the inherent hierarchical relationship between the erroneous syntactic element and their associated correction point; the cells containing “XX” in the table represent impossible selections which will have a probability of zero regardless of the chosen *User Personality Model* parameters.

Table 3-5 - Composition Accuracy Model - Representative `CorrectionPoint` PDF

	Word / Number	Sentence	Paragraph	Document
Word / Number	0.72	0.15	0.05	0.08
Sentence	XX	0.45	0.25	0.30
Paragraph	XX	XX	0.42	0.58

Typing Accuracy Model

The *Typing Accuracy Model* describes a user’s proficiency with the keyboard, in terms of actually striking the key that was intended. The model is probabilistic, as it must describe, for each intended key, the probabilistic distribution of striking that key or any other. For example, the probability that a touch-typist would make an error when intending to strike the letter `‘o’` may be different in the digraph `‘lo’`, where the same finger is likely used to type both letters, than in the digraph `‘mo’` where different fingers are used.

We are not aware of research that details n-graph typing accuracy in the literature, but we argue that it is reasonable to presume that attackers could use such models to characterise a compromised system and we therefore propose the inclusion of such a model in the HGP. The model we propose can be refined for n-graph, as described in [36] and represented as a matrix as shown in Table 3-6 - Typing Accuracy Model - Representative n-graph PDF where each row represents the target n-graph and each column represents the n-graph substituted along with its probability.

Table 3-6 - Typing Accuracy Model - Representative n-graph PDF

	Target / Probability	Candidate 1 / Probability	Candidate 1 / Probability	Candidate 1 / Probability	…
`at`	`at`	`ar`	`af`	`ag`	…
	0.91	0.02	0.03	0.01	…
`ate`	`ate`	`afe`	`atw`	`age`	…
	0.88	0.04	0.02	0.02	…
`The`	`The`	`Teh`	`Thr`	`Rhw`	…
	0.89	0.09	0.01	0.01	…
…	…	…	…	…	…

iii – General Editing Actions Lexicon
When errors are introduced, they must be corrected. The *General Editing Actions Lexicon* represents certain actions that a user takes to edit a `Document`, and it will be detailed during our discussion of Stage 3 at section 3.6.1. It is important to try to minimize the dependencies between aspects of the automatically generated HID behaviour to keep the HGP architecture as simple as possible. We have decided therefore, to separate the concerns associated with the errors on syntactic elements (processed in Stage 3) and the editing actions required to rectify those errors (processed in Stage 4).

The editing actions are processed at Stage 4 and they are described in detail in the *Editing Model*. While we do not do completely define the *Editing Model* here, Stage 3 has a weak dependency on the *General Editing Actions Lexicon* because it must be aware of the following elements from that lexicon: *Move Syntactic Element* (`Move_SE`), where a syntactic element is removed from one location in the `DocMod` and inserted at a different location, and *Replace Syntactic Element* (`Replace_SE`) where one syntactic element is replaced by another.

3.5.2. Transformations
Three passes are required in order to accomplish the transformations necessary to implement Stage 2. The first and second of these passes produces an intermediary DPM while the third produces the stage's output.

Transformation 2.1 – Flag Nodes for Errors
According to the appropriate *User Personality Models*, this transformation stochastically determines which DPM nodes will be flagged for error. For

`Misplaced_SE` the HGP will stochastically determine the number of occurrences of the error that will take place according to the *Composition Accuracy Model* (Table 3-3) and randomly select the appropriate number of syntactic elements for annotation in the DPM. Once the HGP determines that a `Misplaced_SE` error takes place, it must stochastically select the appropriate error details from the *Composition Accuracy Model*.

The HGP will select nodes for `Wrong_SE` by traversing the DPM and examining each node to determine if the target syntactic element or a candidate syntactic element will be selected; choosing a candidate rather than the target results in a `Wrong_SE` error and the candidate syntactic element will represent the error details

Similarly, for each n-graph in its *Typing Accuracy Model* (Table 3-6) the HGP will traverse the DPM node by node to determine which n-graphs, if any, have been mistyped. Should it find that a specific syntactic element has been mistyped; the HGP will use the *Typing Accuracy Model* PDF to select the appropriate error details.

For each `CompError`, the HGP will also stochastically choose the appropriate rectifying actions and `CorrectionPoint`. Finally, this transformation will annotate each error node by adding the following error information to the target syntactic element node as per Table 3-2:

a. An `ErrorID`.
b. The error details.
c. The rectifying actions.
d. The `CorrectionPoint`.

Transformation 2.2 – Insert Correction Points

In this second transformation of Stage 1, the HGP parses the DPM in order to:

1. insert *Correction Nodes* at the appropriate `CorrectionPoint`;
2. insert a `Fix` node under each `CorrectionPoint`. This `Fix` node is used to associate the target syntactic element with the erroneous syntactic element. It contains the `SE_ID` of the target syntactic element and rectifying actions necessary to fix the error; and
3. insert rectifying actions under `Fix` nodes.

Transformation 2.3 – Insert Misplaced Stubs

In the case of `Misplaced_SE`, two nodes are required in the DPM: one must be placed where the erroneous syntactic element is first composed and the other where the target syntactic element is meant to go. We will refer to location where the target syntactic element will ultimately be placed as a `Stub`. The current design allows for movement of a syntactic element by level: `Paragraph` to `Paragraph, Sentence` to `Sentence`, `Word` to `Word` and `Number` to `Number`. This transformation therefore:

1. inserts a `Stub` node in the DPM according to the error's `Position`. This stub will contain the target text, along with the required rectifying actions, and
2. tags the target syntactic element containing the instantiated `Misplaced_SE` and the `Stub` with the corresponding `ErrorID`.

3.5.3. Output: DPM2 – Composition Errors Introduced

The output of Stage 2 is a DPM where select nodes have been annotated with `CompErrors` and where the Insertion Points required to fix those Errors have been included.

3.5.4. Worked Example

Recall the output from Stage 1, which is our starting point for Stage 2 transformations:

```
PARAGRAPH 1 [
SENTENCE 1 [WORD 1 [Here] WORD 2 [is] WORD 3 [a] WORD 4 [great] WORD 5
[sentence]:] ]
PARAGRAPH 2
[SENTENCE 2 [WORD 6 [The] WORD 7 [quick] WORD 8 [brown] WORD 9 [fox]
WORD 10 [jumps] WORD 11 [over] WORD 12 [the] WORD 13 [lazy] WORD 14
[dog].] ]
SIGNATURE [WORD 23 [Sly]]
```

Transformation 2.1 annotates DPM nodes for `CompErrors`. In trying to determine if any `Misplaced_SE` errors are present in our example, the HGP would use Table 3-3 and might determine that there will be two misplaced `Words` ($p = 0.11$) and stochastically chooses `Word 8` and `Word 9`. Let us further presume that the model stochastically chooses to misplace these by four words to the right. The HGP would use the *Composition Accuracy Model* at Table 3-5 to stochastically determine that these errors will be corrected at the `EndOfSentence` (with $p = 0.15$) and annotate the DPM nodes thusly:

```
...[18] WORD 8 [brown CompError (Misplaced_SE WORD 1, Right 4 WORD,
Move_SE (Left 4 WORD), EndOfSentence)] WORD 9 [fox CompError
(Misplaced_SE WORD 2, Right 4 WORD, Move_SE (Left 4 WORD),
EndOfSentence)] ...
```

The HGP would then traverse the DPM and might determine that "great" will be replaced by "good" resulting in a Wrong_SE error according to Table 3-4 (p = 0.08). The HGP would further determine that this error would be noticed and corrected by the synthetic user at the EndOfSentence and DPM node for Word 4 would be annotated as follows:

```
... WORD 4 [great CompError (Wrong_SE WORD 1, good, Replace_SE,
EndOfSentence)] ...
```

While traversing the DPM, the HGP also ascertains if any n-graphs are mistyped according to the *Typing Accuracy Model*. Let us presume that the HGP stochastically determines that the tri-graph "The" is mistyped as "Teh" (p = 0.09 in Table 3-6) and a CorrectionPoint of EndOfWord is chosen (p =0.72 in Table 3-5), resulting in the following annotation to the DPM node:

```
... WORD 6 [The CompError (Mistyped_SE WORD 1, Teh, Replace_SE,
EndOfWord)] ...
```

Transformation 2.2 inserts the appropriate CorrectionPoints in the DPM, along with the appropriate Fix nodes at the end of SENTENCE 1, SENTENCE 2 and WORD 6. This will result in the following changes to the DPM where Fix nodes have been added:

```
... SENTENCE 1 [...WORD 4 [great CompError (Wrong_SE WORD 1, good,
Replace_SE, EndOfSentence) WORD 5 [sentence] : Correction {Fix {Wrong_SE
WORD 1, Replace_SE} } ] ...

... SENTENCE 2 [...WORD 6 [The CompError (Mistyped_SE WORD 1, Teh,
Replace_SE, EndOfWord) Correction { Fix {Mistyped_SE WORD 1,
Replace_SE}]... WORD 14 [dog] . Correction {Fix {Misplaced_SE WORD 1,
Move_SE (Left 4 WORD)} Fix {Misplaced_SE WORD 2, Move_SE (Left 4 WORD)}
} ] ...
```

Finally, Transformation 2.3 inserts Stubs in the DPM where the erroneous text associated with Misplaced_SE error will be entered in the Document. In our example, Transformation 2.1 flagged WORD 8 and WORD 9 for misplacement to

[18] Ellipses are used to indicate information from previous code snippets that is omitted for brevity, while **bold** font is used to highlight additions or changes to the DPM.

the right by 4 Words; the DPM will therefore be updated by adding Stubs as follows:

```
... WORD 7 [quick ...] WORD 8 [brown ...] WORD 9 [fox ...] WORD 10
[jumps] WORD 11 [over] WORD 24 [brown STUB{Misplaced_SE WORD 1, Move_SE
(Left 4 WORD)} ] WORD 25 [fox STUB{Misplaced_SE WORD 1, Move_SE (Left 4
WORD)} ] WORD 12 [the] WORD 13 [lazy] ...
```

3.6. Stage 3 – Editing Action Selection

There are many different ways to create and edit a Document; the synthetic user must be able to make editing choices in a manner consistent with a human user. This stage of the HGP is concerned with selecting the editing actions that are not dependent on the structure of the DocMod or WordProcMod, such as the movement, replacement and typing of syntactic elements.

3.6.1. Dependencies

Editing Model

The synthetic user will carry out a variety of *Editing Actions* to enter text, edit text and navigate the word processor in order to compose the TargetDocument or to edit it. There are many different ways to accomplish editing tasks (such as selecting text and navigating an application interface); the *Editing Model* is a probabilistic model that describes the synthetic user's choices regarding choices of *Editing Actions*.

To the best of our knowledge, no such models have been described in the literature, but it is plausible for attackers to use such a model to characterize the user activity on a compromised system. We have therefore developed a representative *Editing Model* and use it in the HGP. The model details the previously mentioned Move_SE and Replace_SE and many other editing actions. It introduces the deleting of a syntactic element (Delete_SE) where it is removed from the DocMod. Modelled editing actions also include the selection of text where it is highlighted in the word processor (Select_SE), along with copying (Copy_SE) where the selected text is placed on the word processor's clipboard, cutting (Cut_SE) where the selected text is removed from the DocMod and placed on the word processor's clipboard and pasting (Paste_SE) where text from the word processor's clipboard is inserted into the DocMod at the Cursor. Finally, the *Editing Model* provides different methods for positioning the Cursor (Position_Cursor) at a specific location in the DocMod.

Table 3-7 shows a representative *Editing Model* which will contain a discrete PDF of the synthetic user's editing choices. Each row in the table represents one of the possible editing actions listed above, while each column lists the candidate options that could instantiate these editing actions with their probability of being selected. Once again, user personality parameters are necessary to ensure that the *Editing Model* exhibit HID behaviour that is consistent with that of the human user of the compromised system.

It stands to reason that a user's editing actions will be influenced by the word processor used. For example, a user's choice for the Copy_SE editing action would be reduced to two candidate instantiations if the word processor used to compose an email did not provide an Edit *Menu* accessible through the right mouse button.

Table 3-7 - Editing Model

Editing Action	Candidate 1 / Probability	Candidate 2 / Probability	Candidate 3 / Probability	Candidate 4 / Probability
Move_SE	Select_SE, Cut_SE, Position_Cursor , Paste_SE	Select_SE, Mouse Left Button Down, Drag Mouse, Mouse Left Button Up	--	--
	0.75	0.25	XX	XX
Replace_SE	Delete_SE, Type_SE	Select_SE, Paste_SE	Select_SE, Past_SE, Type_SE	--
	0.82	0.05	0.13	XX
Delete_SE	Select_SE, <Delete>[19]	Select_SE, <Backspace>	Position_Cursor, sequence of <Delete>	Position_Cursor , sequence of <Backspace>
	0.40	0.32	0.12	0.16
Select_SE	Move_Mouse, Double-Click	Move_Mouse, Mouse Left Button Down, Mouse Drag, Mouse Left Button Up	Postition_Cursor , <Shift>, sequence of <Navigation Arrows>	--
	0.30	0.45	0.25	XX
Copy_SE	Menu_Area.	<Ctrl-C>	Mouse Right Button	--

[19] We indicate keyboard key names by enclosing them in < > characters.

Editing Action	Candidate 1 / Probability	Candidate 2 / Probability	Candidate 3 / Probability	Candidate 4 / Probability
	Edit_Menu.Copy		Down. Edit_Menu.Copy	
	0.54	0.32	0.14	XX
Cut_SE	Menu_Area. Edit_Menu.Cut	<Ctrl-X>	Mouse Right Button Down. Edit_Menu.Cut	--
	.52	.35	.13	XX
Paste_SE	Menu_Area. Edit_Menu.Paste	<Ctrl-V>	Mouse Right Button Down. Edit_Menu.Paste	--
	.52	.35	.13	XX
Position_ Cursor	Move_Mouse, Mouse Left Button Down, Mouse Left Button Up	Sequence of <Navigation Arrows>	--	--
	.82	.18	XX	XX

The location of the Cursor relative to an error is also likely to influence the choice of editing action; one can surmise that a user who notices an error immediately after mistyping a word is more likely to use <Backspace> than the mouse to delete the word while the reverse might be true if the error is discovered after the entire TargetDocument has been typed. We use the WordProcMod and DocMod respectively to capture the characteristics of the word processor application and Document as it evolves during composition; these are discussed in more detail in section 3.7.1.

The reader will notice however that not all the editing actions listed in Table 3-7 depend on the WordProcMod or DocMod. In order to minimize dependencies between the stages of the HGP, we have chosen to separate the processing of editing actions between those depending on the WordProcMod or DocMod and those that do not. We use the term *General Editing Actions* to mean those editing actions that do not depend on these two models; general editing actions are processed in this stage of the HGP. Conversely, we use the term *Specific Editing Actions* to mean those that do depend on the WordProcMod or DocMod; while those editing actions are processed in Stage 4, this stage of the HGP must be aware of them and there is thus a weak dependency between the *Specific Editing Action Lexicon* and Stage 3 of the HGP.

iii – General Editing Action Lexicon

The *General Editing Action Lexicon* (iii-GenEditActionLex) describes the various Document editing actions that can be chosen by the synthetic user without regard to the structure of the Document or word processor. The editing actions that are rendered at this stage are Move_SE, Replace_SE and Type_SE, as shown inTable 3-8. The complete definition of iii-GenEditActionLex is found at Appendix C.

Table 3-8 – General Editing Action Lexical Examples

Editing Action	Instantiation
Move_SE	Specify a position in the form {Direction, Amplitude}
Replace_SE	One of: Delete_SE, Type_SE, or Select_SE, Paster_SE, or Select_SE, Delete_SE, Paste_SE
Type_SE	Sequence of one or more Characters

The reader will notice that we use a general lexical definition of Move_SE at this stage of the HGP; it will be refined during Stage 4. We chose to design the HGP in this way to more equally distribute complexity between stages 3 and 4.

3.6.2. Transformations

Two passes are required to carry out the Stage 3 transformations. Tranlolsformation 3.1 produces an intermediary DPM and Transformation 3.2 produces the stage’s output.

Transformation 3.1 – Instantiate Terminal Syntactic Elements

Some syntactic elements are collections of others, as is the case with a Sentence that is a collection of Word and/or Number followed by Punctuation. We term those syntactic elements that are not collections to be *Terminal Syntactic Elements* (such as: New_Line, Numbers, Punctuations, Separators, and Words). This transformation tags each of these terminal syntactic elements with a special annotation called Type_SE, which will later be transformed into a sequence of *Keyboard Output Reports* (KOR).

Recall that the DPM has been created from parsing the TargetDocument in Stage 1 and annotated with CompErrors in Stage 2. The HGP must now produce the HID events that will make it appear that the CompErrors took place; it is therefore necessary to annotate the erroneous text, and not the target text, with the Type_SE annotation.

Transformation 3.2 – Instantiate Replace Syntactic Element

Some corrections require the replacement of a syntactic element by another; this action is relatively more complicated than typing and it is thus accomplished with a separate transformation. As mentioned, we chose to carry out this transformation in Stage 3 to distribute the complexity of the HGP. In this transformation, the general editing actions for Replace_SE are instantiated at the appropriate CorrectionPoint by stochastically choosing an appropriate instantiation candidate according to the PDF from Table 3-7.

3.6.3. Output: DPM3 – Editing Actions Selected

The output of Stage 3 is a DPM where nodes have been annotated to reflect the synthetic user's choices with regard to general editing actions that are not dependent on the Document or word processor.

3.6.4. Worked Example

Transformation 3.1 would annotate terminal syntactic elements to show what is to be typed by the synthetic user thusly:

```
... WORD 1 [Here Type {Here} ] ...
```

The reader is reminded that in the case of CompError, it is the erroneous text and not the target text that is annotated as shown here:

```
... WORD 4 [great CompError (Wrong_SE WORD 1, good Type {good},
Replace_SE, EndOfSentence)] ...
... WORD 6 [The CompError (Mistyped_SE WORD 1, Teh Type {Teh},
Replace_SE, EndOfWord)] ...
... WORD 24 [brown Type {brown} STUB{Misplaced_SE WORD 1, Move_SE (Left
4 WORD)} ] WORD 25 [fox Type {fox} STUB{Misplaced_SE WORD 1, Move_SE
(Left 4 WORD)} ] ...
```

Transformation 3.2 is charged with the instantiation of the Replace_SE editing action. From the *Editing Model* (Table 3-7), the HGP stochastically determines that the erroneous Wrong SE WORD 1 (where "good" was used instead of "great") and the Mistyped_SE WORD 1 (where "The" was mistyped as "Teh") will both be instantiated with Delete_SE and Type_SE with a p = 0.82 in each case. This will result in the following Fixes:

```
... SENTENCE 1 [...WORD 5 [sentence] : Correction {Fix {Wrong_SE WORD 1,
Replace_SE {Delete_SE {Wrong_SE WORD 1} WORD 26 [great Type {great}] } }
} ] ...
... SENTENCE 2 [...WORD 6 [The CompError (Mistyped_SE WORD 1, Teh Type
{Teh}, Replace_SE, EndOfWord)] Correction { Fix {Mistyped_SE WORD 1,
Replace_SE {Delete_SE {Mistyped_SE WORD 1} WORD 27 [The Type {The}] } }
} ...
```

3.7. Stage 4 – HID Action Selection

Human users choose what they will say and how they will say it, as ultimately represented by the TargetDocument sent as an email in our operational scenario. As we have seen in our discussion of general editing actions at section 3.6.1, the user's choices for some of those editing actions are not influenced by the environment in which the TargetDocument is being composed. We know however, that users also make choices about editing actions based on the dynamic structure of the Document being composed as well as on the tools available to edit it (the word processor). This stage of the HGP takes the Document and word processor into account to select specific editing actions.

3.7.1. Dependencies

Stage 4 of the HPG depends on the *Editing Model* which has been discussed previously, and it also depends on the DocMod and WordProcMod.

Document Model

The reader will recall from our discussion of i-SynElmtLex at section 3.4.2 that the TargetDocument for which the HID Events are generated must follow a strict structure in order to allow its manipulation by the HGP. We further remember that the HGP uses a *Document Model* (DocMod) in order to keep track of the specific syntactic elements as they are being composed. This is similar to a human user using the screen as a visual feedback aid to make reference to elements of the evolving Document.

As an example, imagine a user omitting the word "College" when typing "Royal Military College of Canada". It is easy to see that the word is omitted when looking at the screen, and a user with such visual feedback is able to see exactly where to position the Cursor in order to insert the missing word. The HGP has no such visual feedback means with which to calculate the intended Cursor location, and it uses the DPM nodes' unique identifiers and a DocMod instead.

The HGP can use different DocMod, but these must generally contain a representation of the Document as it evolves; in fact, each successive transformation uses this DocMod to calculate the distance between the Cursor and syntactic elements of the Document in order to properly instantiate specific editing actions. The layout of a candidate DocMod is shown at Figure 12 and we use this in the design of the HGP.

Our candidate DocMod represents the `Document` as a grid of characters. We chose this structure so that the HGP can translate the relative position of syntactic elements in the DocMod to actual screen coordinates for the generation of `MORs`. Figure 12 suggests that our candidate DocMod uses fixed-width font, but it need not necessarily be so; different mappings between the syntactic element position and their corresponding screen location would simply have to be developed for each variable-width font. The DocMod grid has a fixed width of a certain number of `Columns` (which we set at 80 in this representative DocMod) and an arbitrary number of `Rows` varying between 1 and `Maximum_Rows`.

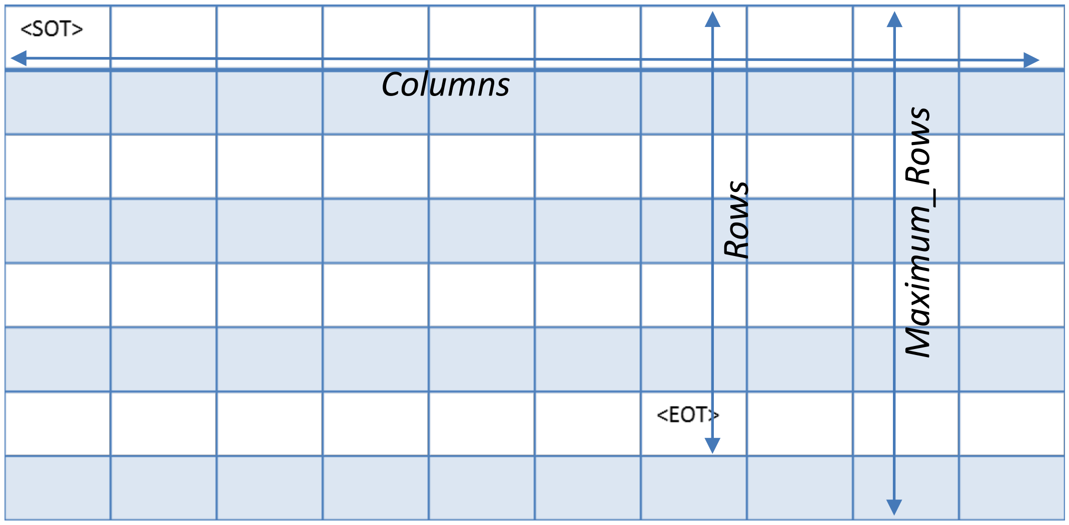


Figure 12 - Candidate Document Model Layout

The DocMod is only ever used by the HGP; it is not meant to be rendered in any human-readable form. That being the case however, the reader can get an idea of its use at Figure 13 which presents the evolving `Document` up to and including the productions required to render `SENTENCE 1` from our worked example.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	…
1	H	e	r	e		i	s		a		g	o	o	d		s	e	n	t	e	n	c	e	:		
2	**EOT**																									
…																										

Figure 13 – Representative Document Model following Sentence 1 Errors

Word Processor Model

The HGP simulates a notional user composing the `TargetDocument` using a specific word processor, which must be modelled by a `WordProcMod`. The `WordProcMod` represents a simple application, such as an email client, where documents can be composed or edited. The HGP can be configured to use a variety of `WordProcMods` by updating the *Editing Model* and `iii-GenEditActionLex` from section 3.5.1 and the *Specific Editing Action Lexicon* below.

In order to design the HGP, we use a `WordProcMod` that makes use of a two-button mouse and a keyboard with a mock-up layout presented at Figure 14. For the design of the HGP, we use a `WordProcMod` having an `Editing_Area` screen

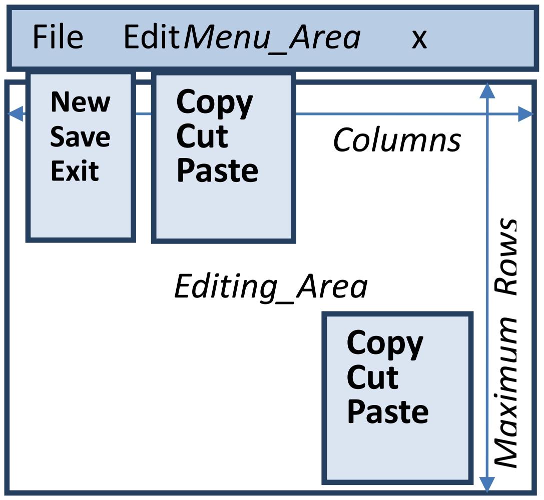


Figure 14 - Representative Word Processor Model Layout Mock-Up

footprint with a width of 800 by a height of 600 pixels; this gives each fixed-width character dimensions of 10 X 12 pixels[20]. The Menu_Area elements will also have defined Screen_Coordinates, but these will remain constant and they will be known a priori.

The WordProcMod contains the elements detailed at Figure 14 which can be grouped in three distinct groups: the *Menu*, the *Editing Section*, *HID Actions* and *Positional Indicators* as shown in Table 3-9.

Table 3-9 - Elements of the Representative Word Processor Model

Menu	
Menu_Area	Menu bar containing positions for the launching of the Edit_Menu, File_Menu and a Close_Button.
Close_Button	Button depicted as an X at Figure 14 which causes the shutdown of the application. It has the same functionality as File_Menu.Exit.
Clipboard	Word processor application buffer used to store a text string.
Edit_Menu	Menu giving choices for the following actions: Copy: copying the selected text to Clipboard. Cut: copying the selected text to the Clipboard and removing it from the Editing_Area. Paste: inserting the text stored in the Clipboard to the Editing_Area, at the Cursor.
File_Menu	Menu giving choices for the following actions: New: Open a new empty Document. Save: Open a save dialog window where the use can choose a file name under which to store the Document currently depicted in the Editing_Area. Exit: Causes shut down of the application.

[20] We are well aware that these this is not a representative application size in typical screen resolutions, but we chose it for illustrative ease of font to screen coordinate mapping.

Table 3-10 - Elements of the Representative Word Processor Model (continued)

Editing Section	
Editing_Area	Section of the application where the Document is displayed and where it can be edited. The area will have a width of Columns, expressed in fixed-width characters, and a height of Maximum_Rows. For this representative WordProcMod, Columns and Maximum_Rows have the same values as defined in the DocMod. The Edit_Menu may also launched by using the right mouse button in the Editing_Area.
HID Actions	
Arrow_Nav	Navigate by moving the Cursor over the Document using the keyboard arrows: Up (UA), Left (LA), Down (DA) and Right (RA)
Double-Click (DC)	Double-click left mouse button.
Drag	Drag the mouse over a portion of the Document while holding the left mouse button.
Left-Click (LC)	Click down the left mouse button.
Left-Release (LR)	Release the left mouse button.
Move Mouse (MM)	Navigate by moving the Pointer over the Document using the mouse.
Right-Click (RC)	Click down the right mouse button.
Right-Release (RR)	Release the left mouse button.
Positional Indicators	
Cursor	Indicator showing where text will be inserted in the Editing_Area.
Pointer	Solid indicator whose position shows the effect of *MM*. Using LC moves the Cursor to the Pointer location in the Editing_Area and activates items in the Menu_Area.

iv – Specific Editing Action Lexicon

Stage 4 is the first stage to consider fully the environment where the TargetDocument is being composed and the *Specific Editing Action Lexicon* (iv-SpecEditActionLex) describes the format of the nodes transformed by this stage of the HGP.

Recall that lexicons define the syntax of the transformation language understood by each stage of the HGP. From Figure 11, the reader notices that Stages 1 through 3 only depended on single *User Personality Model* components which localizes the transformational syntax understood by the respective stage. Stage 4 is different in that it depends on three models: The *Editing Model*, the DocMod and the WordProcMod. The iv-SpecEditActionLex will therefore draw from all three of these models. Although all of these models have been defined previously, we reproduce lexical examples at Table 3-11 for ease of reading; a complete definition of all these elements can be found at Appendix D.

Table 3-11 - Specific Editing Action Lexical Examples

Lexical Element	Definition
Copy_SE	One of the following candidates: Menu_Area.Edit_Menu.Copy, or <Ctrl-C>, or RC.Edit_Menu.Copy
Cut_SE	One of the following candidates: Menu_Area.Edit_Menu.Cut, or <Ctrl-X>, or RC.Edit_Menu.Cut
Delete_SE	One of the following candidates: Select_SE, <DEL>, or Select_SE, <Backspace>, or Position_Cursor, sequence of 1 or more <DEL>, or Position_Cursor, sequence of 1 or more <Backspace>

Move Mouse	One of the following candidates:
	{Left or Right # of Columns, Up or # of Rows}
	{Menu Name.Menu Element}

Move_SE	One of the following candidates: Select_SE, Cut_SE, Position_Cursor, Paste_SE, or Select_SE, LC, Drag, LC.
Paste_SE	One of the following candidates: Menu_Area.Edit_Menu.Paste, or <Ctrl-V>, or RC.Edit_Menu.Paste
Position_Cursor	One of the following candidates: Move_Mouse, LC, LR, or A sequence of 1 or more Nav_Arrows.
Select_SE	One of the following candidates: Move_Mouse, DC, or Move_Mouse, LC, Drag, LR or Position_Cursor, <Shift>, sequence of 1 or more <Nav_Arrows>.

3.7.2. Transformation 4 – Instantiate HID Editing Actions

At this stage in the process, all the editing actions may be instantiated with specific keyboard and mouse HID events, which implies that all the DPM nodes have to be visited. The process will therefore traverse the DPM, instantiating each specific editing action based on the dynamic representation of the `Document`, as contained in the `DocMod` and the probabilistic *Editing Model*, while considering the word processor (as detailed in the `WordProcMod`). Transformation 4 is rather complex, because it accomplishes three main tasks:

1. It traverses the DPM in a temporal order to visit each terminal node,
2. It processes each terminal node by instantiating specific editing actions based on the *Editing Model*, `DocMod` and `WordProcMod`, and
3. It ensures proper management of the `Cursor` when processing `Corrections`.

Visit of DPM Terminal Nodes in Temporal Order

Our chosen design for the HGP has allowed us to treat each DPM node largely independently of other nodes up to this stage. In fact this desire to reduce dependencies, and thus manage complexity, drove our decision to separate the treatment of *general* and *specific* editing actions with the former processes in Stage 3 and the latter processed here. Editing actions must however be done in a logical temporal sequence; we must therefore consider the temporal ordering of editing actions when specifying them because earlier DPM productions will affect later ones. For example, inserting a long rather than a short word in the middle of a `Sentence` will influence how far the mouse would have to be moved to bring the `Cursor` to the insertion point. Furthermore, the HGP must consider the `WordProcMod`, as it will influence the available editing actions; there are more options for moving the `Cursor` in a word processor that uses both a mouse and a keyboard than there are in one that uses a keyboard alone.

Recall that Stage 1 parsed the `TargetDocument` in the same manner that a user reads: starting from top left, moving right along each line and processing lines from top to bottom. Recall from section 3.4.4 that the upper levels of the DPM tree represent the `Document` structure (such as `Paragraphs` and `Sentences`), while the leaves represent editing actions and terminal syntactic elements. The errors and rectifying actions that were introduced in the previous stages used the same structure in the DPM, with the leaves of the tree representing editing actions. A depth first left to right traversal therefore represents an appropriate temporal order for the processing of each DPM terminal node.

Building of the DocMod Representation
As the HGP visits each terminal DPM node in a depth first left to right traversal, it builds the DocMod. The DocMod initially shows an empty Document. When the HPG encounters the first terminal DPM node containing a Type_SE editing action, it places each letter of that syntactic element in successive Document_Positions in the DocMod. Successive editing actions, such as Delete_SE, Move_SE, etc. are similarly reflected in the DocMod (for example, the input strings representing <Backspace> to erase a word and the re-typed characters of that Word modify the associated position in the DocMod). It is important to understand that the DocMod is dynamic, and that its representation at any given time must be associated with the temporal order of the terminal DPM nodes.

Instantiation of Editing Actions
Each DPM terminal node is visited to instantiate specific editing actions. The reader will recall from our discussion of *Transformation 3.1* – Instantiate Terminal Syntactic Elements that the general editing action Type_SE has already been processed. This stage of the HGP can therefore pass those nodes containing Type_SE editing actions without further processing.

When the HGP encounters a terminal node containing an editing action that has not been instantiated, such as Delete_SE for example, it uses the *Editing Model* to stochastically choose a candidate instantiation. Following the example, Delete_SE could be instantiated with Select_SE followed by <Backspace>; the reader will notice that Select_SE can in turn be instantiated by different editing action candidates. The transformation must therefore recursively continue the instantiation of editing actions until the processed node has been instantiated with sequences of either Type_SE or *Mouse Actions*. Type_SE can represent the typing of text or the use of keyboard control characters while *Mouse Actions* will be processed in Stage 5 of the HGP.

Cursor Management during Corrections
Recall that *Transformation 2.2* – Insert Correction Points inserted CorrectionPoints which represent the moment during the composition of the TargetDocument where the synthetic user realizes that there is an error and takes the rectifying actions necessary to Fix it. Recall also that these CorrectionPoints can be at end of each Word, Sentence, Paragraph, or Document. When users realise that there is an error, they must navigate to the erroneous text to effect corrections by moving the Cursor. Once an error is corrected, users must reposition the Cursor to bring it to CorrectionPoint in order to continue with the TargetDocument composition task.

The HGP accomplishes Cursor management based on the *Editing Model*, *Document Model*, and *Word Processor Model*. When processing a DPM node containing a `CorrectionPoint`, the HGP will first save the `Cursor`'s logical position in the DocMod as the point to return to once the rectifying actions have been carried out. The HGP will then obtain the position of the `Cursor` on the screen by using the mapping between its current logical position in the DocMod and its corresponding screen position on the `WordProcMod`. Similarly, the HGP will compute the screen position of the erroneous syntactic element that must be rectified by using a mapping between the DocMod and the `WordProcMod`. The difference between the Cursor's saved location at the `CorrectionPoint` and its required position to carry out rectifying actions will determine which navigation actions (keyboard `ArrowKeys` or *Mouse Actions*) must be carried out to reach the erroneous text. Once the `Cursor` has reached the erroneous text, the rectifying actions can be carried out.

The reader will recall from our discussion of `CorrectionPoints` at section 3.5.2 that the erroneous text contains a reference to the target text. We use the DPM nodes' unique identifiers, for both the target and erroneous syntactic elements, to associate the realization of the error at the `CorrectionPoint` and the rectifying actions at the erroneous text location. Without these associations, it would be impossible for the HGP to determine which instance of a particular syntactic element is erroneous. To illustrate the point, our worked example from section 3.4.5 contained two instances of the `Word` "The", but only the first one was flagged for error.

Following the rectification of an error, the HGP must return the `Cursor` to the position it occupied before the instantiation of the rectifying actions. The HGP will use the mapping between the logical position in the DocMod and the screen position in the `WordProcMod`, for both the corrected text and the saved location of the DPM node associated with the `CorrectionPoint`, to determine the editing actions necessary to return the `Cursor` to its saved location.

3.7.3. Output: DPM4 – HID Actions Selected

The output of Stage 4 is a DPM where nodes have been annotated to reflect the synthetic user's choices with regard to all editing actions.

3.7.4. Worked Example

We continue our worked example by showing how Stage 4 of the DPM chooses specific editing actions for `Delete_SE` and `Move_SE`. The process we illustrate for `Delete_SE` will be very similar for Move_SE which we do not include here. Recall that we left our worked example at the end of Stage 3 in section 3.6.4 having chosen the following general editing actions:

```
... SENTENCE 1 [... WORD 5 [sentence] : Correction {Fix {Wrong_SE WORD
1, Replace_SE {Delete_SE {Wrong_SE WORD 1} WORD 26 [great Type {great}]
} } } ] ...
```

This partial representation of DPM3 shows the first sentence, ending with the word “sentence” and the ‘:’ punctuation mark. Recall that we inserted a `CorrectionPoint` at the end of `SENTENCE 1` during Transformation 2.2 so that the wrongly typed “good” can be deleted and replaced by typing “great”.

When the time ordered traversal of DPM nodes reaches this correction point, the HGP must instantiate `{Delete_SE Wrong_SE WORD 1}` according to the *Editing Model* at Table 3-7. Let us presume that *Candidate 2* was stochastically chosen (p = 0.32) and that the editing action is instantiated with `Select_SE, <Backspace>`, resulting in the following representation of the DPM node:

```
... SENTENCE 1 [... : Correction {Fix {Wrong_SE WORD 1, Replace_SE
{Delete_SE {Select_SE {Wrong_SE WORD 1}} Type <Backspace> } WORD 26
[great Type {great}] } } } ] ...
```

We notice that the node is not yet fully instantiated with `Type_SE` and *Mouse Movements* because `Select_SE` is a general editing action. The HGP uses Table 3-7 once more to choose *Candidate 3* (p = 0.25) and instantiate `Select_SE` as `Position_Cursor`, `<Shift-Down>`, a sequence of `Nav_Arrows` and `<Shift-Up>`. The HGP further determines that the `Cursor` will be positioned at the end of the erroneous syntactic element (“good” in this case) using *Mouse Movements* and it navigates the DPM to obtain the length of the erroneous syntactic element as 4 letters. The HGP can now fully instantiate the specific editing action as follows:

```
... SENTENCE 1 [... : Correction {Fix {Wrong_SE WORD 1, Replace_SE
{Delete_SE {Select_SE {Move_Mouse {End of Wrong_SE WORD 1} Type <Shift-
Down>, Type <LA>, Type <LA>, Type <LA>, Type <LA>, Type <Shift-Up>} Type
<Backspace> } WORD 26 [great Type {great}] } } } ] ...
```

Recall that the HGP depends on the DocMod to instantiate some of the specific editing actions; we reproduce the relevant portions of our worked example from Stage 1 at section 3.4.5 below. We also need the DocMod from Figure 13 which we reproduced at Figure 15 for ease of reading:

```
... SENTENCE 1 [WORD 1 [Here] WORD 2 [is] WORD 3 [a] WORD 4 [great
CompError (Wrong_SE WORD 1, good Type {good}, Replace_SE,
EndOfSentence)] WORD 5 [sentence]: Correction {Fix {Wrong_SE WORD 1,
Replace_SE {Delete_SE {Select_SE {Position_Cursor {End of Wrong_SE WORD
1} Type <Shift-Down>, Type <LA>, Type <LA>, Type <LA>, Type <LA>, Type
<Shift-Up>} Type <Backspace> } WORD 26 [great Type {great}] } } }] ...
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	…
1	H	e	r	e		i	s		a		g	o	o	d		s	e	n	t	e	n	c	e	:		
2	**EOT**																									
…																										

Figure 15 - Representative Document Model following Sentence 1 Errors (from Figure 13)

When processing the `CorrectionPoint` at the end of `Sentence 1`, the `Cursor` is located at `Document_Position` (1, 25)[21], associated with the character space following the colon punctuation character; the HGP saves the `Cursor` location as being associated with that specific character space. The HGP must then traverse the DPM in reverse temporal order to find the target location of the *Mouse Movement* (`End of Wrong_SE WORD 1` in this case). Traversing in reverse temporal order, the HGP first encounters the DPM node associated with the ':' at (1,24). Next the HGP finds the end of `WORD 5` at (1,23) followed by ' ' at (1,15). Finally, the HGP finds the end of `Wrong_SE WORD 1` at (1,14) and deduces that the `Cursor` must be moved left by 10 characters yielding the following DPM node transformation:

```
... SENTENCE 1 [... : Correction {Fix {Wrong_SE WORD 1, Replace_SE
{Delete_SE {Select_SE {Position_Cursor { Move_Mouse {Left 10 characters}
LC, LR }, Type <Shift-Down>, Type <LA>, Type <LA>, Type <LA>, Type <LA>,
Type <Shift-Up>} Type <Backspace> } WORD 26 [great Type {great}] } } } ]
...
```

In summary, the error will be rectified as follows, resulting in the DocMod shown at Figure 16:

1. The mouse will be moved left 10 characters.
2. The left mouse button will be pressed.
3. The left mouse button will be released.
4. A shift key will be pressed.
5. The left arrow key will be pressed four times.
6. The shift key will be released.
7. The backspace key will be pressed.
8. The word "great" will be typed.

[21] `Document_Positions` are expressed as (`Row`, `Column`) in the DocMod.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	…
1	H	e	r	e		i	s		a		g	r	e	a	t		s	e	n	t	e	n	c	e	:	
2	**EOT**																									
…																										

Figure 16 – Representative Document Model following Sentence 1 Corrections

In order to return the `Cursor` to the location at which it was prior to the processing of the `CorrectionPoint`, the HGP computes that the mouse must be moved to right by 10 characters, resulting in the following DPM transformation:

```
... SENTENCE 1 [... : Correction {Fix {Wrong_SE WORD 1, Replace_SE
{Delete_SE {Select_SE {Position_Cursor { Move_Mouse {Left 10
characters}, LC, LR }, Type <Shift-Down>, Type <LA>, Type <LA>, Type
<LA>, Type <LA> , Type <Shift-Up>} Type <Backspace> } WORD 26 [great
Type {great}] } Position_Cursor { Move_Mouse {Right 10 characters}, LC,
LR } } } ] ...
```

3.8. Stage 5 – Mouse Behaviour Processing

In the previous stages, the HGP has processed the use of the keyboard for both the typing of syntactic elements and control characters; Stage 5 aims to bring in the use of the Mouse to the process. This stage will model the use of the mouse in terms of movement speed and button use, and it will translate the `Document_Postions` used in Stage 4 into `Screen_Coordinates`.

Human users are not perfect in their use of HID, which implies that HGP must generate mouse actions in a manner that is consistent with humans making mistakes. This stage of the process introduces *Mouse Errors* (keyboard errors have been introduced in *Stage 2* – Composition Error Introduction).

3.8.1. Dependencies

v – Mouse Behaviour Lexicon

The *Mouse Behaviour Lexicon* (v-MouseBehaviourLex) has two main components: the synthetic user's characteristic use of the mouse in terms of speed and direction, and the potential errors that the synthetic uses can make when using the mouse. Those errors are based on the synthetic user's ability to hit the desired `Screen_Coordinates` and the ability to carry out `Button_Actions` as intended.

The v-MouseBehaviourLex will represent *Mouse Movements* (`MouseMove`) by indicating target `Screen_Coordinates`[22] and speed (`MouseSpeed`) in pixel/s. The v-MouseBehaviourLex must be able to represent *Mouse Errors* (`MouseErrors`) and we chose to model three of them: missing a target location at which a mouse movement is aimed (`Missed_Loc`), the accidental pressing of a mouse button (`Accidental_But`) and the pressing of a wrong mouse button (`Wrong_But`).

The modelling of mouse button errors uses a discrete PDF in a manner similar to that used to model composition errors in section 3.5.1. In the case of `MouseErrors`, the DPM nodes will contain the following information:

1. The *Target* - the intended mouse action.
2. The *Error* – the erroneous mouse actions.
3. The *Rectifying Actions* – the sequence of Editing Actions required in order to transform the erroneous mouse action into the target mouse action.

In contrast to `CompErrors`, it is not necessary to specify a `CorrectionPoint` for `MouseErrors`. We assume that `MouseErrors` will always be corrected immediately after being made because a user making use of a mouse is looking at the screen and will therefore not delay in noticing the error. This lack of a `CorrectionPoint` also means that it is not necessary to give the `MouseError` a unique error identifier. There is no need to refer between the erroneous mouse action and the target mouse action because the rectifying actions are immediately carried out; this is explained in more detail in our discussion of Transformation 5.1 – Specify Mouse Errors and Corrections at section 3.8.2. The complete v-MouseBehaviourLex can be found at Appendix E but lexical examples showing `MouseErrors` can be found at Table 3-13.

Mouse Model

The *Mouse Model* (`MouseMod`) describes the synthetic user's ability to hit the desired `Screen_Coordinates` and to carry out `Button_Actions` as intended. It has two main components: the synthetic user's characteristic use of the mouse (in terms of speed and direction) and the potential errors that the synthetic uses can make when using the mouse.

---

[22] `Screen_Coordinates` are expressed in pixels as (X, Y) where X is the horizontal coordinate component representing `Columns` and increasing to the right and where Y is the vertical component increasing downward and representing `Rows;` the origin is located in the top-left corner of the screen.

The *Mouse Model* (MouseMod) is inspired by the work of Traore and Ahmed [36]. From their work, we are interested in the elements of Table 2-3 that deal with movement direction (Mouse_Direction) and speed (Mouse_Speed). Specifically, we consider *Movement Speed compared to Distanced Travelled* (MDS) and *Average Movement Speed per Movement Direction* (MDA). The *Average Movement Speed per Types of Action* (ATA) will be considered in Stage 7.

Table 3-12 - Mouse Behaviour Lexical Examples

Movement			
Move_Mouse	Target Screen_Coordinate (X, Y)	MouseSpeed (pixel/s)	
MouseErrors	**Target Mouse Action**	**Erroneous Mouse Action**	**Rectifying Action**
Accident_But	NULL	One of: RC, RR or LC, LR	One of: LC, LR or <Esc>
Missed_Loc	(X, Y)	(X, Y)	(X, Y), Mouse_Speed
Wrong_But	One of: RC, RR or LC, LR or DC	One of: RC, RR or LC, LR or DC	One of: LC, LR or LC, LR, DC or <Esc>, DC or NULL

Our MouseMod also uses an integer from 1 to 8 to designate the octet in which the movement is taking place. It should be noted however that where Traore and Ahmed's first octet began at 0° from the top of the screen, ours begins at -22.5° from the vertical (337.5°), as depicted at Figure 17. We chose this to ensure that

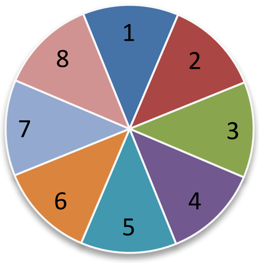


Figure 17 - Mouse Model Movement Direction Octets

mouse movement targets that are horizontal or vertical compared to the Pointer's starting location fall within the centre of a specific octet. In other words, moving the mouse horizontally will result in movement in the 3rd or 7th octet while moving it vertically will result in movements in the 1st or 5th octet.

In order to model movement speed, the MouseMod will first consider the movement direction of travel to pick normally distributed average speed, similar to Traore and Ahmed's MDA measure. This movement speed will then be adjusted based on the distance that must be travelled, considering Traore and Ahmed's MSD. While we have not conducted experimentation to populate the MouseMod our intuition is that movements over small distances will be slower than those over large distances because they require finer motor control; work on Fitts Law such as [50] could help refine this stage. Table 3-13 provides a representative movement speed specification for a potential MouseMod used in the development of the HGP.

Table 3-13 - Mouse Model – Representative Movement Speed Statistical Distributions

	Movement Direction Octet				Movement Speed Reduction Factor (%)			
Octet # Angle (°)	**1 (-22.5 – 22.5]**	**2 (22.5 - 47.5]**	**…**	**8 (-47.5 - - 22.5]**	**000 - 125**	**125 - 250**	**…**	**≥ 700**
Mean (µ) (pixel/sec)	220	231	…	90	32	29	…	0
Standard Deviation (σ) (pixel/sec)	12	15	…	11	--	--	--	--

As discussed in the previous section, the MouseMod must also model the mouse button errors made by the syntactic user, namely Accident_But and Wrong_But. While this model has not been populated, representative models were used to develop the HGP. Table 3-14 shows the distribution on the number of mouse button errors made in the composition of the TargetDocument, while Table 3-15 shows a representative discrete PDF of the rectifying actions that would fix errors in the use of the mouse buttons. Accident_But is the clicking of a button when none was intended during a Move_Mouse; it is expected that this error can be rectified by either clicking the left mouse button (with LC, LR) or by simply continuing the Move_Mouse. According to our WordProcMod, accidently pressing the right button when the left was intended (either singly or during a DC) causes the Edit_Menu to open; we see two options to close the menu and the model offers two candidates: pressing the <Esc> key or pressing the left mouse button twice. Lastly, accidently pressing the left mouse button when the right button was intended is simply fixed by pressing the right button.

Table 3-14- Mouse Model – Representative Number of Mouse Errors PDF

Number of Mouse Button Errors	0	1	2	...
Accident_But	0.62	0.18	0.09	...
Wrong_But	0.82	0.09	0.05	...
Number of Missed_Loc Errors	0	1	2	...
Probability	0.25	0.42	0.18	...

Table 3-15 - Mouse Model – Representative Mouse Button Errors Rectifying Action PDF

	Target	Error	Rectifying Actions Candidate 1	Rectifying Actions Candidate 2
Accident_But	Move_Mouse	Move_Mouse, RC, RR, Move_Mouse	Move_Mouse, RC, RR, LC, LR, Move_Mouse	Move_Mouse, RC, RR, <Esc>, Move_Mouse
		0.45	0.62	0.38
	Move_Mouse	Move_Mouse, LC, LR, Move_Mouse	Move_Mouse, LC, LR, Move_Mouse	--
		0.55	1.00	XX
Wrong_But	RC, RR	LC, LR	RC, RR	NULL
		1.00	1.00	XX
	LC, LR	RC, RR	LC, LR	<Esc>, LC
		1.00	0.62	0.38
	DC	RC, RR	LC, LR, DC	<Esc>, DC
		0.35	0.52	0.48
	DC	LC, LR	DC	NULL
		0.65	1.00	XX

Finally, the MouseMod must model the Missed_Loc error. In order to develop the HGP, we used a model of Move_Mouse accuracy based on the grid shown at Figure 18. The target Document_Position is found at the centre of the grid (denoted by T in the figure) and potential erroneous locations are numbered in a clockwise spiral around the target. A representative MouseMod mouse accuracy discrete PDF giving the probability of hitting the target Document_Position or an erroneous location can be found at Table 3-16.

24	9	10	11	12
23	8	1	2	13
22	7	T	3	14
21	6	5	4	15
20	19	18	17	16

Figure 18 - Mouse Model – Move_Mouse Accuracy Layout

Table 3-16 - Mouse Model - Representative Move_Mouse Accuracy PDF

	Target	Candidate 1	Candidate 2	...	Candidate 24
Document_Position	0.68	0.03	0.12		0.01

To recap, we have discussed the following aspects of the MouseMod: the movement direction octets, a PDF to determine stochastically the speed of movement based on that direction and including an adjustment for the distance of travel, PDFs to select stochastically the number of MouseErrors, PDFs to instantiate stochastically MouseErrors in terms of buttons and location. We are now ready to discuss the two specific transformations being carried out by Stage 5 of the HGP.

3.8.2. Transformation 5.1 – Specify Mouse Errors and Corrections

The HGP first annotates the appropriate DMP nodes with MouseErrors According to the *Mouse Model* (Table 3-14) this transformation stochastically determines how many MouseErrors will take place. Once the number of errors has been determined, the HGP will randomly select which mouse actions will be flagged as erroneous and annotate the corresponding DPM node with the following information:

1. The *Target* - the intended mouse action which is already noted in the DPM node.
2. The *Error* – the erroneous mouse actions. The HGP will use the MouseMod (Table 3-15) to instantiate mouse button errors and to pick erroneous Document_Positions (Table 3-16).
3. The Rectifying Actions – the sequence of editing actions required in order to correct the Mouse_Error according to the MouseMod (Table 3-15).

Recall that MouseErrors rectifying actions take place immediately after the error, and that a CorrectionPoint need not be specified.

3.8.3. Transformation 5.2 – Compute Mouse Movement Details

At the conclusion of the previous stage, DPM4 contains Move_Mouse of the form: Move_Mouse {Right 10 characters, Down 2 rows} or Move_Mouse {Edit_Menu.Copy}; these must now be translated to Screen_Coordinates in order to compute the Pointer trajectory in Stage 6.

Recall from our discussion of the WordProcMod that our chosen word processor application has a defined screen dimension expressed in pixels. The Menu_Area elements are located at fixed screen locations, and the HGP holds a list of their constant Screen_Coordinates; it need not compute them as it does for Move_Mouse actions that are expressed relative to the Cursor. Each character in the Editing_Area has a Document_Position that is 10 pixels wide by 12 pixels high; the mapping between Document_Position and Screen_Coordinates is therefore given by:

$$(X, Y) = (Column * 10, Row * 12) \qquad (1)$$

The Speed must also be specified to detail Move_Mouse actions, but we leave its specification to Stage 6.

3.8.4. Output: DPM5 – Mouse Errors Introduced

The output of Stage 5 is a DPM where nodes have been annotated to reflect all of the synthetic user's choices with regard to all editing actions, including the typing of syntactic elements, errors and their rectifying actions. The terminal DPM nodes are all annotated with Type_SE, Move_Mouse or mouse button actions.

3.8.5. Worked Example

Recall the following portion of our worked example from Stage 4:

```
... SENTENCE 1 [... : Correction {Fix {Wrong_SE WORD 1, Replace_SE
{Delete_SE {Select_SE {Postion_Cursor { Move_Mouse {Left 10 characters},
LC, LR }, Type <Shift-Down>, Type <LA>, Type <LA>, Type <LA>, Type <LA>
, Type <Shift-Up>} Type <Backspace> } WORD 26 [great Type {great}] }
Postion_Cursor { Move_Mouse {Right 10 characters}, LC, LR } } } ] ...
```

In carrying out *Transformation 5.1*, The HGP will first use the DocMod at Table 3-14 to determine that there will be one instance of an Accident_But (p = 0.18), no instances of Wrong_But (p = 0.82) and one instance of Missed_Loc (p = 0.42). Let us further presume that both DPM nodes containing the Move_Mouse editing action above (in italics) have been stochastically selected as erroneous mouse actions to implement the Accident_But and Missed_Loc errors respectively.

The HGP uses Table 3-14 to determine that the Accident_But is to be instantiated with an accidental press and release of the right mouse button (p = 0.55). Such an error happens in the middle of a mouse movement, and the associated Move_Mouse is split into two sections: one before and one after the button error. In this case, the HGP determines that the error will take place at the fourth character of the movement, leaving a 6-character movement after the error. The HGP further determines that *Rectifying Actions Candidate 2* will be used (p = 0.38) and the error will be corrected with the use of the <Esc> key. The first Move_Mouse DPM node is therefore transformed as follows:

```
... Postion_Cursor { Move_Mouse {Left 4 characters}, RC, RR, Type <Esc>,
Move_Mouse {Left 6 characters}, LC, LR }...
```

In order to instantiate the Missed_Loc error, the HGP uses Table 3-16 to determine that the error will be instantiated with *Candidate 4* (p = 0.04). From Figure 18, we recall that this error position is 1 Row below and 1 Column to the right of the target. The HGP therefore transforms the second Move_Mosue DPM node as follows:

```
... Postion_Cursor { Move_Mouse {Down 1 row, Right 11 characters},
Move_Mouse {Up 1 row, Left 1 character}, LC, LR } ...
```

Transformation 5.2 traverses the DPM and finalises the details of the Move_Mouse errors. To illustrate the technique, we use the last Move_Mouse action discussed above: Move_Mouse {Down 1 row, Right 11 characters}. Recall that the HGP builds a dynamic DocMod as it traverses the DPM when arriving at this node (Figure 15 at this point). From this representation of the DocMod, the HGP can ascertain that the Cursor position is at Column 16, Row 1; going right 11 characters and down 1 row puts the target of the mouse movement at Column 27, Row 2. Using formula (1), we transform this into$(27 *$

$10,2 * 12) = (270,24)$. The `Move_Mouse` action is therefore transformed as follows:

```
... Move_Mouse {(270, 24), }, ...
```

3.9. Stage 6 – HID Event Stream Generation

From the point of view of the attackers' presence on the compromised computer system, a synthetic user generates a series of HID Events on the USB. Stage 6 of the process traverses the DPM and transforms `Type_SE`, `Move_Mouse` and mouse button editing actions into a sequence of *Keyboard Output Reports* (KOR) and *Mouse Output Reports* (MOR).

3.9.1. Dependency: vi – HID Event Lexicon

The *HID Event Lexicon* (`vi-HIDEventLex`) describes the HID events that are created by the HGP to render the TargetDocument on the compromised system. It defines the two HID events of interest: MOR and KOR as defined in [51]. The definitions for these elements can be found at Appendix F.

3.9.2. Transformation 6 - Generate KOR and MOR

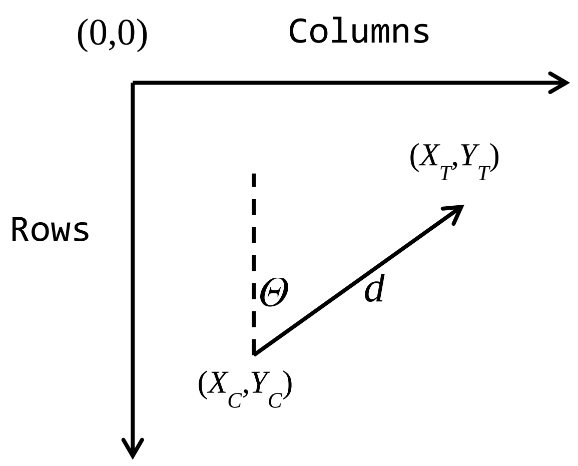
As was done in Stage 4, the HGP will perform a temporal order traversal of the terminal DPM nodes, building the DocMod as it goes. The HGP will instantiate `Type_SE` editing actions by generating one KOR for each letter in a syntactic element and for each control character (used for editing or navigation) in the DPM. Similarly, the HGP will instantiate the mouse button actions (LC, LR, RC, RR and DC) by generating an MOR for each.

The HGP must calculate the trajectory of the `Pointer`, in terms of `Screen_Coordinates` and distance, for each `Move_Mouse` action. Recall from Table 3-13 that the *Mouse Model* details the movement speed depending on the distance to be travelled by direction. Given the *current* `Screen_Coordinates` of the `Pointer` (X_C, Y_C) and its *target* `Screen_Coordinates` (X_T, Y_T), the distance (*d*) can be computed using equation (2) while the angle (*Θ*) can be computed using equation (3):

$$d = \sqrt{(X_T - X_C)^2 + (Y_T - Y_C)^2} \qquad (2)$$

$$\theta = \operatorname{atan2}\big((X_T - X_C), (Y_C - T_T)\big) * 180/\pi \qquad (3)$$

The *arctan* trigonometric function requires adjustments to the angle depending on the quadrant of the verctor between the current and target `Screen_Coordinates`; we use *atan2* [52] instead of *arctan* to obtain the appropriate angle regardless of the quadrant. We also invert the function



arguments from (*Y*, *X*) to (*X*, *Y*) to obtain angles relative to the *Y*-axis which corresponds to the top of the screen, as demonstrated at Figure 19.

Figure 19 - Mouse_Move Trajectory Calculations

The HGP can use the *Mouse Model* at Table 3-13 to populate the `Move_Mouse` with the stochastically chosen `Speed`, according to the `Direction` octet and adjusted for the `Distance`.

3.9.3. Worked Example

Recall that the first word of our `TargetDocument` is:

```
... WORD 1 [Here Type {Here} ] ...
```

When transforming this node, the HGP will create a KOR for each letter (as detailed at Appendix F) as follows:

- KOR with flag 0x05 to indicate that the right <Shift> key is depressed and keycode 0x0B for the letter ‘h’,
- KOR for keycode 0x08 for the letter ‘e’,
- KOR for keycode 0x15 for the letter ‘r’, and
- KOR for keycode 0x08 for the letter ‘e’.

Recall the following DPM node from the Stage 5 worked example in section 3.8.5:

```
... Move_Mouse {(270, 24), }, ...
```

The HGP will use the `WordProcMod` to keep track of the position of the `Pointer`. It is the OS that keeps track of the position of the `Pointer` but that position is available to applications such as a word processor; we therefore keep track of its position within the `WordProcMod`. Let us presume that the previous mouse action had left the `Pointer` 20 characters left and 10 rows below the `Cursor`'s current `Screen_Coordinates` at (270,24). Using equation (1), we can derive the `Pointer`'s current position at

$$(270 - 20 * 10, 24 + 10 * 12) = (70,144) \qquad (4)$$

We can now use equations (2) and (3) respectively to compute the distance and angle of the trajectory between the current and target `Pointer` position as follows:

$$d = \sqrt{(270 - 70)^2 + (24 - 144)^2} = 233\, pixels \qquad (5)$$

$$\theta = \text{atan2}\big((144 - 24), (270 - 70)\big) * 180/\pi = 40.0° \ (6)$$

Having computed the trajectory of the `Mouse_Move`, the HGP is now in a position to calculate the movement speed using Table 3-13 - Mouse Model – Representative Movement Speed Statistical Distributions as follows:

1. *Θ* = 40.0° places the movement in octet #2 which implies a movement speed of 233 pixels/s.
2. *d* = 233 implies a reduction factor of 29% giving a movement speed of $233 * 0.29 = 165$ pixels/s, giving the following transformation to the DPM node:

```
... Move_Mouse {(270, 24), 165}, ...
```

3.9.4. Output: DPM6 – HID Events Generated

The Stage 6 DPM is a tree representing a sequence of `MOR`s and `KOR`s as its terminal nodes.

3.10. Stage 7 – Event Timing Characterization

As discussed in section 2.4, the timing of HID events can be used to characterize users. The HGP must therefore generate HID events at a rate that is consistent with generation by a human. This stage of the HGP assigns delays between the HID events.

3.10.1. Dependencies

HID Timing Model

The *HID Timing Model* is a probabilistic model describing the distribution of times between the occurrences of various HID events. The model considers the timing of keyboard keys, mouse buttons and the transition between MORs and KOR. While we have not conducted experiments to populate the *HID Timing Model*, we propose a representative continuous PDF of the delays between HID elements as shown at Table 3-17.

Table 3-17 - HIT Timing Model - Representative PDF

	KOR Delays				MOR - Delays							
	aa	ab	…	zz	LC-LR	LR-RC	…	DC	KOR – LC	KOR – RC	MM – KOR	…
Mean (μ) (pixel/sec)	342	112	…	458	.75	.88	…	.152	1.560	2.123	3.158	…
Standard Deviation (σ) (pixel/sec)	27	11	…	32	.05	.11	--	.20	.45	.55	.78	…

As we have done for the *Typing Accuracy Model*, we propose to model digraphs. The left hand section of Table 3-17 shows the digraphs with the delay between each letter. Similarly, the *HIM Timing Model* considers the delays between various mouse button actions and transitions between MORs and KORs as shown in the right-hand section of the table.

vi – Timed HID Lexicon

The *Timed Event Lexicon* (`vii-TimedEventLex`) defined at Appendix G simply describes ordered pairs of each HID Event and their release time on the USB.

3.10.2. Transformation 7 – Insert Inter-Report Delays

This stage traverses the DPM in temporal order and inserts release times for each HID Event, for both MORs and KORs.

DPM-6 contains the `Move_Mouse` editing actions containing a target `Screen_Coordinate` and movement speed of the form:

```
... Move_Mouse {(270, 24), 165}, ...
```

Transformation 7 uses the movement `Speed` contained in the DPM node to generate MORs at a rate of 200 per second, which is typical for wired HID mice [53]. This value reporting rate is established between the USB device and the endpoint during USB enumeration. From our previous example where 233 pixels were to be travelled at a speed of 165 pixel/s, the HGP will generate:

$$233\ pixels * \frac{1}{165}\frac{seconds}{pixel} * 200\frac{KORs}{second} = 282\ KORs \qquad (7)$$

The HGP will therefore generate 282 KORs on the trajectory between the current and target `Pointer` position with a delay of $^{1}/_{200}$ second between each.

The delays between other HID events, such as keyboard keys, mouse buttons and transitions between KORs and MORs will follow the *HID Timing Model* as illustrated at Table 3-17.

3.10.3.Final Output

The final output of the HGP is a sequence of HID Events rendering the composition of the `TargetDocument` on the compromised system, in accordance with the *User Personality Model*.

3.11. Chapter Summary and Conclusion:

This chapter has presented the HGP, which is the major contribution of the research. The process accepts a `TargetDocument` containing structured English free-text, and through a series of transformations, produces a sequence of HID Events that, when placed on the USB of the compromised system, gives attackers the impressions that the `TargetDocument` is being composed by a human user.

Recall from section 1.6 that the aim of the research is *to develop a conceptual framework for the automatic generation of HID events in a manner that is consistent with a human inputting text into a computer system*. The process presented here meets this aim because it is able to automatically generate HID events for placement on the USB. The process helps resolve the specific deficiencies addressed in section 1.5 by automatically generating the `MORs` and `KORs` that correspond to the composition of English free-text by a synthetic user.

The pipes and filters architecture, with which the process described in this chapter has been designed, offers flexibility. The dependencies between the *User Personality Model* components and the transformation stages of the process are localized and well defined. This architecture will enable the extension of the process by allowing for the use of enhanced models of user behaviour and refined transformation stages to represent defenders' evolving understanding of attacker capabilities.

3.12. Consistency with Human Behaviour

The process that we have described in this chapter is clearly able to automatically generate HID Events. But that is not sufficient however, for us to claim success. Our aim also requires us to show that process supports the generation of HID that are consistent with a human user composing text on the compromised system. We argue that the chosen architecture is conducive to the inclusion of a varied

User Personality Model; a more complete argument for the validity of the User Personality Model is presented at section 5.4.2. As defenders refine their understanding of the models that attackers use to characterise compromised computer systems, they are in a position to refine the *User Personality Model* used in the process.

CHAPTER 4 : FEASIBILITY THROUGH IMPLEMENTATION

4.1. Chapter Introduction

We believe that the HGP framework presented in this research is a valid solution to the problem of automatic generation of HID events, in a manner that is consistent with a human user; we have devoted Chapter 3 to making this validity argument. Proposing a valid solution is not sufficient however, to demonstrate that this research meets its aim. The solution presented here must also be feasible, and this chapter will be devoted to making such a feasibility argument.

This chapter will discuss the proof-of-concept requirements and the implementation decisions made to meet those requirements. The chapter will then discuss the implementation of a proof-of-concept of the HGP, including some of the specific technologies and programming languages used. Finally, the chapter will conclude by arguing for the sufficiency of the proof-of-concept as implemented toward a demonstration of feasibility.

4.2. Proof-of-Concept Requirements

In order to demonstrate that the HGP described in Chapter 3 is a feasible solution to the real and important problem described in section 1.5, the author has built a proof-of-concept system, called the *Synthetic User Environment* or SUE. Demonstrating that the HGP can be built is proof of its feasibility, ipso facto. We must demonstrate however that the SUE, as it was built, is a sufficient implementation of the HGP. To that end, we propose the following proof-of-concept requirements:

1. The proof-of-concept must be able to accept `TargetDocuments` containing arbitrary English language free-text as its input. The structure of the English free-text must respect the `i-SynElmtLex`, but there are no semantic restrictions on its composition.
2. In order to respect the pipes and filters architecture that we have selected for the HGP, each successive stage of the proof-of-concept must be able to produce an output that is an acceptable input to its follow-on stage.
3. To meet the research aim's automatic generation component, each stage of the proof-of-concept must be able to automatically manipulate the nodes of the DPM. This is not taken to mean that the proof-of-concept must implement all transformations; it must be demonstrated however that all transformations can be implemented automatically. This requirement for automation implies:

   a. That the proof-of-concept must be able to manipulate a `DocMod` to handle the dynamic nature of the `Document`.
   b. That the proof-of-concept must be able to use an appropriate `WordProcMod`.
4. The proof-of-concept must demonstrate that it is possible to consider and integrate aspects of the *User Personality Model*, including the processing of errors and corrections.
5. The final stage of the proof-of-concept must produce HID events that can be placed on the USB of compromised system, along with inter-event delays, such that these HID events must render the composition of the `TargetDocument`, including errors and their correction.

These requirements capture the essential difficulties in the implementation of the HGP. We further suggest that meeting those requirements will give assurance of the feasibility of the HGP as designed in the paragraphs that follow.

We have chosen to implement those aspects of the HGP that model the use of the keyboard by the synthetic user, and our proof-of-concept SUE application generates the `KORs` associated with the rendering of the `TargetDocument` as it is composed on the compromised system. We do not however, implement the use of a mouse in the SUE.

Our decision to only model the use of the keyboard is partly due to equipment limitations. As we will explain in more detail in our discussion of the SUE, our proof-of-concept SUE uses a PLX Net2280 USB Development Board [54]. Each PLX Net2280 can only represent one USB device, and only one can be installed on a PC. Furthermore, only one such PLX Net2280 was available to the author, which forced us to represent only one of the two types of HID devices specified by the HGP. Of the two devices considered by the HGP, the keyboard is definitely the most important and we will discuss why we believe that it suffices to demonstrate the feasibility of the HGP in section 4.4.

The implementing `KORs` exercises the significant portions of the HGP, and that it meets the requirements listed above. Specifically, the implementation of the keyboard aspects of the HGP allows the proof-of-concept to accept an English free-text `TargetDocument` as required by 1 above.

The editing actions described in the *User Personality Model* contain many candidate instantiations that use the keyboard; `KORs` are therefore sufficient to demonstrate that the proof-of-concept can handle the processing of errors and corrections. We have chosen to implement a simplified version of the *Editing Model* which instantiates specific editing action candidates. We have also implemented a simplified version of the *HID Timing Model* which stochastically inserts delays between KORs according to parameters representing the typing proficiency of the synthetic user. We argue that the integration of the *Editing Model* and of the *HID Timing Model* demonstrate that the *User Personality Models* can be integrated in the proof-of-concept, thereby meeting the requirement listed at 4 above.

In order to meet the requirement listed at 3 above, we have chosen to implement all of the automatic processing for transformations associated with keyboard HID events in stages 1, 3, 4, 6 and 7. We have also implemented automatic processing of transformations 2.2 and 2.3 in Stage 2. We deem the transformations automatic in that they manipulate the DPM that they receive from the previous stage, and pass their resulting output DPM to the follow-on stage as required by 2 above until the last stage generates a series of KORs that meet the requirement of 5 above.

4.3. Proof-of-Concept Development - The SUE

The SUE proof-of-concept was implemented using the TXL and C programming languages. TXL development took place using an open source editor which supports TXL syntax highlighting, called Notepad++ [55], under the Windows XP [56] and Windows 7 [57] OS. C development was accomplished with the Eclipse IDE for C/C++ Developers [58].

Because it is less widely used than C, we begin this section with a discussion of the TXL programming language, followed by a description of the implementation of the various stages of the HGP in the SUE.

The TXL Programming Language

TXL is a product of Queen's University [59]. It is a programming language specifically designed to support transformational programming, which has made it particularly well suited to the pipes and filters architecture chosen for the HGP, where each stage's input is transformed into the stage's output according to the
where each stage's input is transformed into the stage's output according to the
transformation rules described in Chapter 3. As depicted in Figure 20, included here by permission, a TXL program is composed of three phases: *Parse*, *Transform* and *Unparse* [60]. Each of these will be discussed in the context of the SUE.

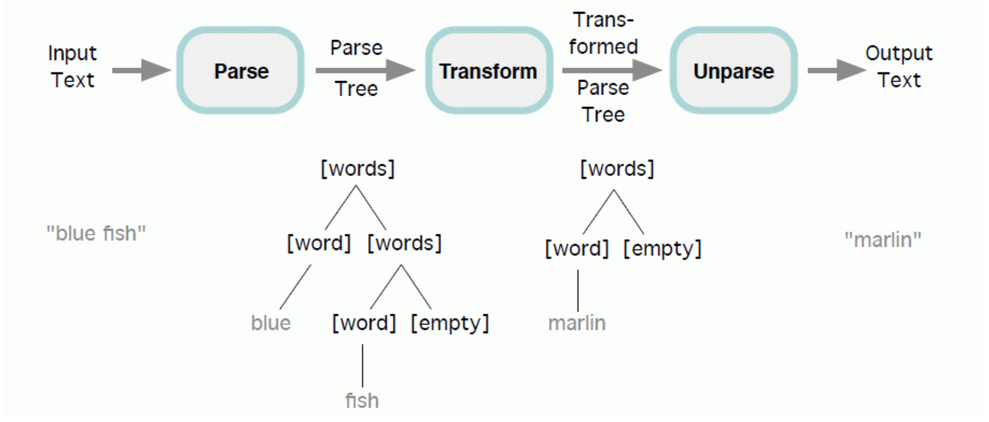


Figure 20 - The Three Phases of TXL [60]

The TXL *Parse* Phase parses the input of the TXL program into a tree. Parsing is done according to the appropriate grammar [61], which is represented by the various lexicons discussed in Chapter 3. The grammar is expressed in a notation similar to *Extended Backus-Nauer Form* [62], and it must be context-free. For the SUE, the transformation stages where TXL is used (in stages 1 through 3) utilise the lexicons which describe how the various model elements are structured in order to parse the stage's input.

The TXL *Transform* Phase takes care of transforming the input parse tree into an output parse tree according to the rules and functions specified in the program. The transformations are specified *by example* meaning that a *pattern* and *replacement* is specified for each; if a tree matches the *pattern*, it is replaced by the *replacement*. Every TXL function and rule produces a node that is homomorphic to its input, meaning that both are of the same type. In the SUE, the TXL Transform Phase is used to implement the HGP transformations described in sections 3.4 to 3.7, with the exception of *Transformation 2.1 – Flag Nodes for Errors*.

The TXL *Unparse* Phase therefore produces an output that can also be parsed by the grammar; in our case this means that each stage's lexicon must define both the input and output DPM elements syntax. In our case, this means that every successive lexicon using TXL is an extension of its predecessor. For example, recall the first sentence from the `TargetDocument` from our worked example, which became the following nodes in DPM-1:

```
Here is a great sentence: ... →

... SENTENCE 1 [WORD 1 [Here] WORD 2 [is] WORD 3 [a] WORD 4 [great] WORD
5 [sentence]:] ...
```

The first stage's lexicon (`i-SynElmtLex`) therefore defines a `Word` as both `[wordToken]`, to parse the input syntactic element, and also as `'WORD [number] '[ [wordToken] ']` to reflect the fact that words are tagged and numbered by Stage 1.

TXL manipulates text files, and our implementation of stages 1 through 3 contain 25 TXL files. Figure 21 gives an indication of the relationship between the four elements of a TXL transformation in the context of the SUE:

1. The *input* file. This is the `TargetDocument` for the first transformation and the output of a TXL program for subsequent transformations. The input file's extension is the name of the TXL program (`TargetText.1_SynWElmtExtracton` above),
2. The *TXL program* which effects the transformations; TXL program files have an extension of `.txl` (`1_SynElmtExtraction.txl` above),
3. The *lexicon* which represents the grammar used by the TXL program to parse its input. By convention, we use the `.lexicon` file extension (`SynElmt.lexicon` above) , and
4. The *output* file which is a DPM in the context of the SUE. The output file extension is the name of the follow-on stage's TXL program (`DPM1.2-1_CompErrorFlagging` above).

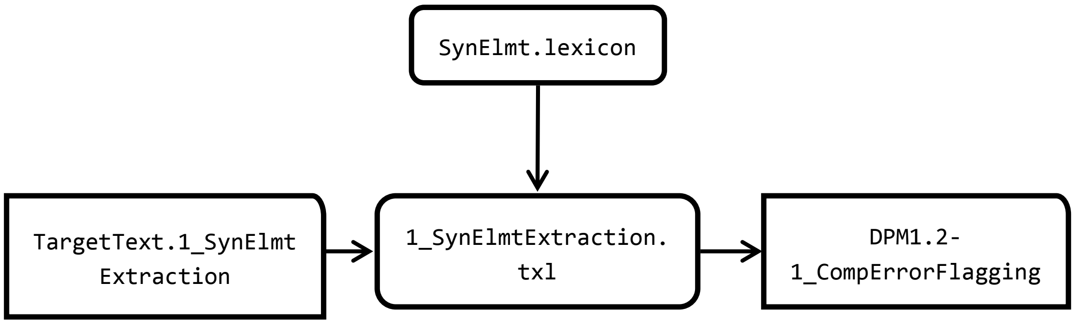


Figure 21 - TXL Software Architecture in the Context of the SUE

4.3.1. Implementation of Stage 1

The input of this stage is a `TargetDocument` that contains English free-text in the sense that there are no restrictions on the English language text that is being composed by the synthetic user; there are however restrictions on its structure. It must be a UTF-8 encoded text document that respects the structure laid out in the TXL implementation of `i-SynElmtLex`; this lexicon is implemented as a TXL grammar. The implementation recognizes:

- `Word` - a sequence of alphabetic characters (censored swearing such as "d$%$it?" would not be recognized as a `Word`),
- `Number` - as an optional '-', optional '$', integer or real number beginning with a digit and continuing with any number of digits, an optional decimal point followed by at least one more digit, and an optional exponent beginning with the letter E or e and followed by an optional sign and at least one digit [60]; the following would all be recognized as `Numbers`: "-$454.27", "0.08%", "-4.6E-3",
- `Sentence` – sequence of `Word` or `Number` followed by a `Punctuation` mark,
- `Paragraph` – a sequence of `Sentences` followed by a `New_Line`, and
- `Signature` – a sequence of `Words` followed by a `New_Line`.

In *Transformation 1* – Identification of Syntactic Elements, the `TargetDocument` is parsed such that each `Word`, `Number`, `Paragraph` and `Signature` is uniquely identified and structured in a tree with a single `Document` node at its root and instances of `Word`, `Number`, `New_Line` and `Punctuation` nodes at its leaves. `DPM1.2-1_CompErrorFlagging` contains this output DPM. By default, DPMs are output as UTF-8 encoded text files. TXL can create XML output, which has proven useful for debugging, but there is no need for human interpretation of the DPMs.

The result of this stage, DPM1, closely resembles the example we discussed at section 3.4.5. For example, a `TargetDocument` such as:

```
Here is a great sentence:
The quick brown fox jumps over the lazy dog.
Sly
```

results in the following DPM-1:

```
PARAGRAPH 1 [
SENTENCE 1 [WORD 1 [Here] WORD 2 [is] WORD 3 [a] WORD 4 [great] WORD 5
[sentence]:]]
PARAGRAPH 2 [
SENTENCE 2 [WORD 6 [The] WORD 7 [quick] WORD 8 [brown] WORD 9 [fox] WORD
10 [jumps] WORD 11 [over] WORD 12 [the] WORD 13 [lazy] WORD 14 [dog].] ]
SIGNATURE [WORD 23 [Sly]]
```

4.3.2. Implementation of Stage 2

Stage 2 of the process introduces Composition Errors. *Transformation 2.1* – Flag Nodes for Errors which flags nodes for errors. This transformation has not been automated because it is heavily dependent on the *Composition Model* and *Typing Accuracy Model*; we argue that these models must be developed further before representations of these models can be manipulated by the SUE. The nodes are flagged for error manually in a manner consistent with was is expected by a TXL implementation and following the stage's grammar: *ii – Composition Error Lexicon*. That is to say that each syntactic element flagged for error is annotated with the required information from the lexicon as detailed in section 3.5.2. From our worked example in section 3.5.4, the reader will recall the `Word` "the" was mistyped as "teh", this would result in the following manual annotation:

```
... WORD 6 [The CompError (Mistyped_SE WORD 1, Teh, Replace_SE,
EndOfWord)] ...
```

The resulting intermediary DPM2-1 is fed to the *Transformation 2.2* – Insert Correction Points implementation which inserts the appropriate `CorrectionPoints` for each syntactic element flagged as erroneous. Based on the `CorrectionPoint` identified in the `CompError`, the transformation finds the appropriate location in the DPM and creates the `Correction` node in the DPM if one does not exist. The transformation then inserts a `Fix` node that contains the identification of the erroneous syntactic element and the appropriate rectifying actions. For `Mistyped_SE` error from our example, the `Correction` is inserted as follows:

```
...WORD 6 [The CompError (Mistyped_SE WORD 1, Teh, Replace_SE,
EndOfWord) Correction { Fix {Mistyped_SE WORD 1, Replace_SE}]...
```

The intermediary DPM2-2 is in turn fed to *Transformation 2.3* – Insert Misplaced Stubs which inserts `Stubs` to allow the processing of `Misplaced_SE` errors. Whenever such a `Misplaced_SE` error is encountered, the SUE navigates the DPM according to the error's `Position`, and inserts a copy of the erroneous syntactic element so that it is typed out of position during composition. Recall that in our example, "brown" was annotated for misplacement by four `Words` to the right, resulting in the following insertion of `Word 24` in the DPM.

```
... WORD 7 [quick ...] WORD 8 [brown CompError (Misplaced_SE WORD 1,
Right 4 WORD, Move_SE (Left 4 WORD), EndOfSentence)] WORD 9 [fox ...]
```

```
WORD 10 [jumps] WORD 11 [over] WORD 24 [brown STUB{Misplaced_SE WORD 1,
Move_SE (Left 4 WORD)} ] ...
```

4.3.3. Implementation of Stage 3

Stage 3 of the process selects those Editing Actions that are not dependent on the Document or word processor. While the design of Stage 3 of the process called for two transformations, the TXL implementation further refines those into five passes.

Transformation 3.1 – Instantiate Terminal Syntactic Elements instantiates terminal syntactic elements, which are those syntactic elements that are not collections of syntactic elements such as `New_Line`, `Numbers`, `Punctuations`, `Separators`, and `Words`. This transformation tags each of these terminal syntactic elements with a special annotation so that they can later be transformed into a sequence of KOR.

This transformation is implemented in two passes. The first pass TXL program traverses the DPM using a depth first left to right ordering and annotates terminal syntactic elements that were not flagged for error. From our worked example, `Word 1` would be annotated as follows:

```
... WORD 1 [Here Type {Here} ] ...
```

Recall that those nodes that have been flagged for error contain both the target text and the erroneous text. The TXL program will annotate those nodes flagged for error such that KORs will later be generated from their erroneous text. Our mistyped `Word` would be annotated as follows:

```
...WORD 6 [The CompError (Mistyped_SE WORD 1, Type{Teh}, Replace_SE,
EndOfWord) Correction { Fix {Mistyped_SE WORD 1, Replace_SE }]...
```

The second and third passes are TXL programs that implement *Transformation 3.2* – Instantiate Replace Syntactic Element and instantiate `Replace_SE` and `Move_SE` errors respectively. Our chosen implementation does not implement the full *Editing Model* discussed in chapter but it does demonstrate that the model can be integrated in the HGP by automatically generating the general editing actions associated with composition errors. For example, the implementation of the specialisation of the `Replace_SE` general editing action associated with the previously discussed erroneous `Word` is given here:

```
...WORD 6 [The CompError (Mistyped_SE WORD 1, Type{Teh}, Replace_SE,
EndOfWord) Correction { Fix {Mistyped_SE WORD 1, Replace_SE {Delete_SE
{Mistyped_SE WORD 1} WORD 27 [The Type {The}] } }]...
```

TXL was found to be sub-optimal for the implementation of stochastic processes, as required by the proposed *User Personality Model*. The SUE is therefore implemented such that DPM3 is the last to make use of TXL. The fourth and fifth passes in Stage 3 are TXL programs that annotate nodes containing specific editing actions so that the further stages of the process will be able to process them. The fourth pass annotates the sequences of `Fix` nodes in correction points to enable `Cursor` management in the next stage and the fifth pass extracts all of the terminal nodes of the DPM (either `Type` or specific editing actions) into a file for processing by the next stage. The output of this stage, DPM3, is a text file that will be parsed by the follow-on C application.

4.3.4. Implementation of Stage 4

Transformation 4 – Instantiate HID Editing Actions is implemented in C using 15 files grouped into six modules. Recall that Transformation 4 – Instantiate HID Editing Actions visited the nodes of the DPM in temporal order to choose specific editing actions for each general editing action it encountered. At this stage of the HGP, the choice of editing action depends on the structure of the `Document` as well as the word processor in use. The program implementing this stage must also keep track of the position of the `Cursor` so that it may be able to handle corrections by navigating to erroneous DPM nodes and back.

The program uses two data structures to model the `Document` (DocMod) and the sequence of HID Events (`HIDSeq`) that is used to compose the TargetDocument on the compromised system. These two data structures are used for different purposes: the DocMod is used to compute relative distances between syntactic elements and the `HIDSeq` will ultimately be the output of the stage: DPM4. Let us address each of these in turn.

We implement the DocMod as a linked-list where each list node contains a terminal syntactic element identifier, its text and length. Recall from our discussion of the `Document` Model that the Document was represented as grid to allow for the calculation of relative distances between syntactic elements in the correction of errors. As the proof-of-concept implementation of the HGP only deals with the HID events associated with a keyboard, it suffices to represent this distance as relative `Document_Positions` without having to compute translations to `Screen_Coordinates`. The SUE therefore computes relative positions by traversing the DocMod linked-list and counting the number of characters associated with each syntactic element. Recall that a `CorrectionPoint` represents a point in the composition of the `Document` where the synthetic user realises an error has been made, and that the `Cursor` records the composition position in the `Document`. The program uses a pointer to refer to the Cursor in the DocMod linked-list.

The HIDSeq on the other hand is a list of all the keyboard actions that will become HID events. It is important to note that the HIDSeq is rate monotonic, regardless of what is being rendered in the word processor. Typing the character 'a' and erasing immediately with the <Backspace> key still results in two KORs. The program uses a custom built parser to interpret the nodes of DPM3, and consumes syntactic elements by using a depth-first, left-to-right traversal of the Document tree.

As mentioned in our discussion of the Word Processor Model, some of the synthetic user's choices with regard to HID actions depend on the word processor being used; using Microsoft Word is very different than using vi. For the purposes of our proof-of-concept, we have chosen to use the Microsoft Notepad application that is part of the Windows OS [56] as the WordProcMod. The specific editing actions possible with Notepad are therefore integrated as the WordProcMod into the C implementation of *Transformation 4* – Instantiate HID Editing Actions.

For each Syntactic Element, HID actions are processed as follows:

1. Type_SE actions are implemented by adding a syntactic element to the DocMod. In those cases where a syntactic element is inserted in the middle of other text, a linked-list node is inserted appropriately.
2. Delete_SE actions have multiple possible instantiations as proposed in the *Editing Model*; in order to demonstrate the feasibility of the approach, the Delete_SE action is implemented by adding a sequence of <Backspace> to the HIDSeq.
3. Move_SE is implemented with Select_SE, Cut_SE, Position_Cursor, Paste_SE
4. Select_SE is implemented with a sequence of <Shift> <Arrow_Keys>
5. Cut_SE is implemented with <Ctl-X>
6. Position_Cursor is implemented with a sequence of <Arrow_Keys>
7. Paste_SE is implemented with <Ctl-V>

The output of this stage, DPM4, contains a sequence of all HID events representing the composition of the TargetDocument by the synthetic user using keyboard events.

4.3.5. Combined Implementation of Stages 6 and 7

Transformation 6 - Generate KOR and MOR is charged with the generation of HID events based on the HIDSeq represented by the previous stage of the HGP. We chose to use a USB development board, the PLX Net2280, installed on a SUE

Server, in order to generate HID events to be placed on another workstation representing the compromised system as shown at Figure 22.

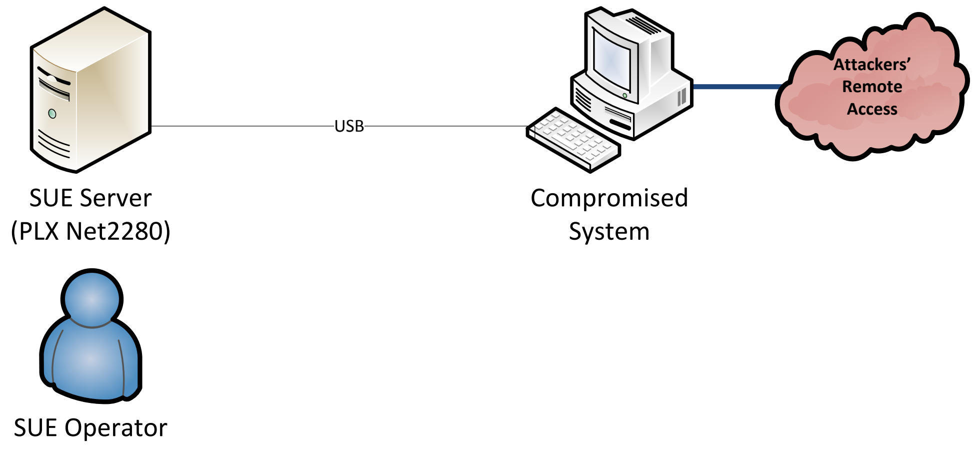


Figure 22 - Physical Setup of the SUE Proof-of-Concept

The PLX Net2280 is a PCI to Hi-Speed USB 2.0 programmable peripheral controller [54]. The SUE applications are run on a server equipped with a PLX Net2280. This server is connected to the compromised computer system with a USB cable where it enumerates as a keyboard. There are no connections between the SUE server and the compromised system, other than a USB cable.

As mentioned previously, we only had access to one PLX Net2280 for this research. The reader should also note that only one instance of a PLX Net2280 may be used on a host at one time; the PLX Net2280 API cannot manage more than one USB device. Both of these facts drove our decision to implement only the keyboard aspects of the HGP. Stages 6 and 7 are implemented together in a C application that makes use of a module representing the *HID Timing Model* and also uses the PLX Net2280 API.

Transformation 6 - Generate KOR and MOR of the HGP is responsible for the generation of HID events; as our proof-of-concept SUE only implements the aspects of text composition using the keyboard, it generates KORs only. In order to generate KORs, the SUE parses the HIDSeq to extract the characters that correspond to the typing of syntactic elements and control characters corresponding to editing actions. The SUE uses the PLX Net2280 API to craft the appropriate KORs from those characters in order to insert them onto the USB of the compromised system. From our worked example, the SUE would generate four KORs for WORD 1 [Here Type {Here}] as follows:

1. KOR for ‘H’
2. KOR for ‘e’
3. KOR for ‘r’
4. KOR for ‘e’

The last transformation of the HGP, *Transformation 7* – Insert Inter-Report Delays, introduces delays between the HID events according to the *HID Timing Model*. Because the proof-of-concept SUE is only concerned with KORs, we need only consider those aspects of *HID Timing Model* that deal with `KORs`. We chose to implement this portion of the *HID Timing Model* with a normal distribution parameterised according to the synthetic user’s typing proficiency. The last program in the SUE obtains the synthetic user’s typing proficiency parameter from the SUE Operator and releases the `KORs` generated at *Transformation 6* - Generate KOR and MOR on the USB of the compromised system with the stochastically modelled delays.

The proof-of-concept SUE application is used by running a series of pipelined command-line programs. These programs can be easily scripted which further goes to demonstrate that the HGP described at Chapter 3 can be automated.

4.4. Sufficiency

A full implementation of the HGP as described in Chapter 3 would certainly demonstrate the feasibility of the proposed framework, but it is considered to be out of scope for this research. We argue that such a complete implementation is also not necessary to demonstrate the feasibility of the HGP, and believe that the proof-of-concept SUE as implemented is sufficient. This section is therefore devoted to this sufficiency argument and it will do so by discussing how the proof-of-concept requirements have been met and why the omitted portions are not deemed essential.

The SUE’s implementation meets our first proof-of-concept requirement because it is demonstrated to be able to accept an English free-text `TargetDocument` contained in a text file. The SUE automatically extracts the syntactic elements from that `TargetDocument` and outputs a DPM representation of its syntactic elements. The SUE accepts any English free-text `TargetDocument` as long as it respects the Stage 1 lexicon at Appendix A.

DPMs are successively passed through 10 TXL transformations in Stages 1, 2 and 3 and passed through five further transformations programmed in C in Stages 4, 6 and 7. These successive transformations demonstrate that the chosen pipes and filters architecture is valid and that the HGP can be automated as demanded by requirements 2 and 3.

The SUE achieves the third proof-of-concept requirement by implementing portions of the *Editing Model* and *HID Timing Model*. The integrations of these two models shows that the HGP as designed can integrate *User Personality Model* components that include stochastic candidate choices of model elements, and it has shown that it can use those models to automatically transform DPMs. This meets the fourth proof-of-concept requirement.

The SUE is able to render the composition of the `TargetDocument`, including errors and their corrections, on the compromised system with only a USB connection between the SUE Server and the compromised system. This addresses the last proof-of-concept requirement, ipso facto.

We recognize that some elements of the HGP are not implemented in the proof-of-concept SUE. The three major omissions are the integration of all aspects of the *User Personality Model*, the decision not to generate `MORs` and the manual annotations of errors in Transformations 2.1. We leave the discussion of the validity *User Personality Model* to section 5.4.2 and discuss the other two omissions below.

Equipment limitations prevented us from implementing those aspects of the HGP that deal with the generation of `MORs`. In the context of text composition, the mouse is used to navigate the application and the OS; the mouse is therefore useful in the carrying out of editing actions but it cannot be used to enter text. Because of this, we consider that the mouse is subsumed by the keyboard; while many users are likely to often prefer using a mouse to a keyboard to navigate, there is nothing that can be done with a mouse that cannot be done with a keyboard (within the context of Notepad as our word processor). It should also be noted that the PLX Net2280 API includes primitives that can move the mouse at a given speed to specified `Screen_Coordinates`; the implementation of `MORs` is therefore likely to be straight-forward if time consuming. We argue therefore, that the proof-of-concept SUE need not implement `MORs` to demonstrate that the HGP is feasible.

Our decision to manually flag nodes for error, instead of implementing their automatic annotation, in *Transformation 2.1* – Flag Nodes for Errors could be questioned but we argue that this decision does not invalidate our feasibility argument. Flagging nodes for error is unique in that it relies on two components of the *User Personality Model*, namely the *Typing Accuracy Model* and *Composition Accuracy Model*. As we have discussed in section 3.10.1, the *HID Timing Model* and the *Typing Accuracy Model* are very similar in that they model the interactions between character digraphs. We have successfully implemented stochastic delays, parameterized according to characteristics of the synthetic user, in the SUE as discussed in the . The implementation of a digraph discrete PDF as prescribed by the proposed *Typing Accuracy Model* could be accomplished by following a similar approach.

As we will argue further in section 5.4.2, the *Composition Accuracy Model* is one of two components of the *User Personality Model* for which the least information is available in the literature. This means therefore, that our efforts to implement a representative approximation of that model would be very speculative, and we suggest that it would contribute little to the validation of this research. Nonetheless, we can examine what would be required to implement the automatic flagging of errors at *Transformation 2.1* – Flag Nodes for Errors:

1. Develop a representation of the *Typing Accuracy Model* showing a discrete PDF of digraph candidates similar to what is shown in Table 3-6.
2. Develop a representation of the *Composition Accuracy Model*'s various elements as shown in the following: Table 3-3, Table 3-4 and Table 3-5.
3. Develop an application that can parse DPM1 from the previous stage and extract syntactic elements. Recall that TXL does not handle stochastic processes well, which means that this step must be done without the power of a transformation oriented language.
4. Stochastically choose nodes for errors according to the *Composition Accuracy Model* and *Typing Accuracy Model*, and annotate those nodes accordingly.

We argue that the steps above are straight forward and well understood. The automation of *Transformation 2.1* – Flag Nodes for Errors is not necessarily technically challenging, but it would be time consuming and onerous. We do not believe that this effort is required to demonstrate the feasibility of the HGP and are confident that it can be safely omitted from the proof-of-concept implementation.

4.5. Chapter Conclusion

The proof-of-concept SUE has been implemented to demonstrate the feasibility of the HGP described in Chapter 3. The proof-of-concept SUE was implemented using two programming languages, TXL and C, on Windows XP or Windows 7 OS. The proof-of-concept SUE is meant to be executed on a server equipped with a PLX Net2280 USB development board connected to the compromised computer system via USB, where it will enumerate and appear as a keyboard.

Stages 1 through 4, 6 and 7 of the HGP have been implemented for the generation of KORs and this chapter has described their implementation. The chosen pipes and filters architecture is respected and the proof-of-concept SUE is demonstrated able to render a `TargetDocument`, including errors and corrections, on a compromised system. The successful implementation of the proof-of-concept SUE demonstrates that the solution presented in Chapter 3 is feasible.

CHAPTER 5 : SUMMARY AND CONCLUSION

5.1. Chapter Introduction

This research tackles the problem of intelligence collection on sophisticated attackers who have compromised a computer system. This chapter summarises the dissertation and explain why we believe that we have met our research aim. We first argue that the problem we tackled, namely the collection of intelligence on sophisticated attackers is real and important. We then present our solution to this problem, explain why it is valid and argue for its feasibility. The chapter then lists the contributions of our work and discuss avenues of future work.

5.2. Deficiencies of Current Approaches

The *Remove Clean Restore* response to the discovery of the presence of attackers on a computer system is inadequate because it is reactive; attackers have the initiative and defenders are always left trying to catch up. In order to have better chances of developing effective defences, defenders must know more about the attackers tools, techniques, capabilities and motives. Intelligence gathering is therefore required on the attackers and *Network Counter-Surveillance Operations* (NCSO) has been proposed as a means of collecting such intelligence.

We surmise that sophisticated attackers who put significant effort and resources into the compromise of high-value targets will examine the systems they compromise closely to derive valuable information from them and to be assured that they are not themselves under observation. Attacker observation of the high-value systems that they compromise can include characterization of users, which implies that sophisticated attackers can be interested in the ways with which users interact with the systems they have compromised, including through observation of the HID events on the USB.

We have presented an operational scenario describing the compromise of a high-value computer system by sophisticated attackers, which would warrant the collection of intelligence. We argue that attackers would want their activity on that compromised system to remain undetected because of the value they derive from presence on the high-value compromised system; this will affect what they can do to characterize user activity. Current techniques do not provide a means of collecting intelligence on sophisticated attackers in such a context.

5.2.1. Honeypots

We have discussed honeypots in section 2.2. While honeypots can be beneficial for capturing tools and techniques, they do not allow for the collection of intelligence on sophisticated attackers because attackers are likely to easily discover that the honeypot they have compromised is of low-value. Honeypots are defined as having no production value, and sophisticated attackers are not likely to be fooled by a system that does not correspond to the production systems that they target.

Current research does not detail user activity on honeypots; we believe that this makes honeypots unrealistic intelligence collection mechanisms from the point of view of a sophisticated attacker. The lack of HID activity on the system will be another avenue through which sophisticated attackers can discover that they have been led to a honeypot instead of a high-value system.

Finally, honeypots are static; the research literature does not provide means of changing the information on a honeypot to respond to an evolving situation. This makes honeypots unsuitable for setting the stage for future operations against the attackers (which we termed the *preparation* of the attackers).

5.2.2. NIST

The typical response to compromise breaks contact with attackers and thus does not allow defenders to collect intelligence on the attackers, as would be the case in a *Network Counter-Surveillance Operation* (NCSO). There are serious deficiencies in the tools and techniques that defenders currently have at their disposal in order to allow intelligence collection on attackers, and we have described a *Network Intelligence Surveillance Toolset* (NIST) to address these deficiencies. The NIST is premised on maintaining contact with the attackers by allowing them to remain on the compromised system while they are being surreptitiously observed. While provisions are being made for the control of the attackers' actions on the compromised system, the NIST carry an inherent amount of risk and should only be engaged when the assessment of the risk against the potential intelligence gathered warrants an NCSO.

The major deficiency addressed by this research deals with the fact that attackers are likely to break contact themselves if they suspect that they are under observation. For an NCSO to be successful, the NIST employed by network defenders must provide the attackers a realistic environment with which to interact. It is this specific aspect of the NIST that is being tackled by this research.

5.3. Aim of the Research

From section 1.6, the aim of our research is:

To develop a conceptual framework for the automatic generation of HID events in a manner that, when observed by attackers, is consistent with a human inputting text into a computer system.

The *generation of HID events* must take place outside the compromised system to ensure that it is not visible to the attackers. *Automatic* means that a human user must not use interface devices to generate HID events on the compromised system's USB. *Consistent with a human inputting text* means that the rendering of the document composition will be characterized as normal by the attacker as depicted at Figure 2. This consistency with humans further means that the rendered composition must respect local HID event stream semantics which in turn implies that replaying recorded HID events on the compromised system's USB would not suffice.

Our literature review and the discussion of NCSOs and NISTs allow us to say that the problem tackled by this research is real and that solving it will bring valuable contributions to the field of computer network defence.

5.4. Validation Argument

In order to demonstrate that the HGP that we have propose is valid, we must successfully argue that it is a sufficient solution to the problem and that it meets our research aim. We must also show that the *User Personality Model* used by the HGP as represented in this work is representative of the models that may be used by attackers to characterize human activity on a compromised system. Finally, we must explain why we believe that the proof-of-concept SUE demonstrates the feasibility of the HGP.

5.4.1. Sufficiency

The *HID Event Generation Process* (HGP) that we have designed in this research proposes an actual solution to the research problem. The HGP accepts a free-text English language `TargetDocument` and produces a series of HID events that render its composition on a compromised computer system. The HGP is systematic in its approach, and it uses a sound pipes and filters architecture that is conducive to modification and expansion as our understanding of the attackers' means of user characterisation evolves. The HGP's generation of user events is automated because there is no need for humans to use HID to generate HID events. The HGP successively transforms the `TargetDocument` based on a defined *User Personality Model* which can be parameterised which allows for the representation of different synthetic users.

Specifically, the HGP generates HID events related to the use of a keyboard and mouse, such that the syntactic elements of the `TargetDocument` will be rendered as they are being composed on the compromised system as follows:

1. The first stage of the HGP extracts all syntactic elements, such `Word`, `Numbers`, `Sentences`, `Paragraphs`, etc. from the TargetDocument and produces a *Document Production Model* (DPM) using a tree to represent the structure of the syntactic elements of the `TargetDocument`.
2. Stage 2 of the HGP annotates the syntactic elements of the DPM with *Composition Errors*, which are mistakes in the typing, choice or positioning of syntactic elements. The Composition Errors introduced at this stage are not associated with the structure of the `Document` or the word processor being used. In order to ensure consistency with human behaviour, as required by our aim, this stage of the HGP uses two components of the *User Personality Model*, namely the *Composition Accuracy Model* describing the synthetic user's propensity to misplace or to use the wrong syntactic element, along with the *Typing Accuracy Model* that details the synthetic user's propensity to mistype syntactic elements.
3. The third stage of the HGP selects the editing actions that are required to render the typing of the DPM's syntactic elements, along with the introduced errors and associated corrections. To maintain consistency with human behaviour, the HGP uses an *Editing Model* that describes the synthetic user's preferences in the selection of editing actions. The editing actions selected and described in Stage 3 are general in that they do not depend on the representation of the `Document` or word processor being used.
4. In Stage 4, the HGP refines the choices of editing actions based on the word processor being used. It also considers the dynamic representation of the `Document` as it is being composed and the *Editing Model*, in order to choose specific editing actions in a manner consistent with the synthetic user.
5. The fifth stage of the HGP considers the use of a mouse. The stage picks screen coordinate targets for the implementation of specific editing actions and computes trajectories (distance and angle) and movement speed according to the *Mouse Model* to maintain consistency with a human user. The stage also introduces errors in the use of the mouse, along with the corrections of these errors. Namely, it models the accidental pressing of a mouse button, the pressing of the wrong mouse button and the missing of a target location on the screen.

6. Stage 6 generates HID events, namely *Mouse Output Reports* and *Keyboard Output Reports* to implement the editing actions of the DPM in accordance with the word processor being used.
7. Finally, the last stage of the HGP introduces delays between the HID events in accordance with the *HID Timing Model*.

We believe that the HGP summarized above, and described in detail in Chapter 3, is a sufficient solution to the problem of the provision of a realistic user environment at the HID level for the purpose of intelligence collection on sophisticated attackers as detailed in our research aim.

5.4.2. Validity of the User Personality Model

In order to successfully argue that the aim of the research has been met, we must show that the automatically generated HID events are *consistent with those generated by a human* inputting text into a computer system. Recall from Figure 2 that attackers characterize a system by comparing the activity they sample against their models of normal behaviour. Unfortunately, attackers are loath to share their models with defenders.

Our review of the open literature has revealed that attackers are willing to expand significant effort to characterise honeypots. One can surmise therefore, that sophisticated attackers who compromise high-value computer systems (such as described in the *Operational Scenario*) would expend significant effort and resources to characterise their compromised target. Attackers can use user activity at the HID level on the USB to characterise the users of the compromised system. Attackers have access to publicly available literature, and we argue that they can use this literature to model user behaviour at the interface level. Defenders and attackers will be engaged in a measure counter-measure arms race in the context of NCSO. While our research has not revealed any such attacker models of user personality in the open literature, it is reasonable to expect that this is something that can be done by sophisticated attackers.

We have discussed modelling efforts for the use of a mouse and keyboard. As mentioned, we have not populated these models through experimentation because this effort was deemed to be out of scope for our research. We have however developed the components of the *User Personality Model* as discussed in Chapter 3. The following paragraph will argue for the validity of these components.

The *HID Timing Model* describes the rate at which our synthetic user is generating HID events on the compromised system's USB. The modeling of typing behaviour based on digraph duration in this model flows directly from the research efforts of

Gunneti and Picardi at [42]. The approach used by Gunneti and Picardy of recording the duration of digraphs could be extended to the use of mouse buttons, and we suggest that it is reasonable to suggest that attackers would model the speed of mouse button action in such a manner. Recall that it is not necessary to model the rate at which MORs is placed on the USB because it is established during the enumeration of the USB device. We can therefore be confident that our *HID Timing Model* is valid as it is closely related to research efforts in this area.

Similarly, many aspects of the *Mouse Model* can found in the open literature. The mouse movement modeling flows from the efforts of Ahmed and Traore at [39] who introduce the concept of speed per movement direction (through their MDA measure) and average movement speed per direction (MSD). The concept of local HID event semantics that we have introduced is based on the DocMod; the mouse movements that will be used to manipulate the Document during its composition will therefore also respect the distribution of movements according to directions, as represented by the MDH measure because they are based on those semantics. The other aspects of the *Mouse Model*, namely the mouse button errors and the mouse use accuracy, are not well documented in the literature. We suggest however, that these are feasible aspects of mouse behaviour that can be modelled by attackers. An attacker could model this aspect of user behaviour through experimentation that counts the number of erroneous mouse actions; we argue therefore that our approach of modelling the accuracy of button use through a discrete PDF is reasonable.

The *Typing Accuracy Model* which we use to model the mistyping of a syntactic element (Mistyped_SE) is also based on Gunetti and Picardi's work at [42]. While the authors only dealt with the issue of n-graph duration statistics, we argue that their treatment of typing based on n-graphs can be extended to accuracy. Attackers who are interested in the accuracy of a HID user will need a means of characterising this accuracy. The approach of determining the number of errors for each digraph variation is one that attackers could reasonably be expected to use to model this typing accuracy, and it is well captured by the *Typing Accuracy Model*'s use of a discrete PDF to represent the candidate digraphs that can be typed by the synthetic user.

Our proposed *Composition Accuracy Model* models two classes of errors that can be made in the composition of a Document: the misplacing of text (Misplaced_SE) and poor choices in the selection of a syntactic element (Wrong_SE). The model also determines at which point the synthetic user realises that an error has been made. We believe that these three aspects, along with

`Mistyped_SE` errors are representative of the types of composition errors that can be made by a user.

Finally, the *Editing Model* instantiates editing actions such as: text selection, moving, replacement, deleting, copying, cutting and pasting. Because users have more than one option for carrying out these editing actions, the model is probabilistic and contains a PDF of the possible candidate instantiations. The model depends on the representations of the word processor used by the synthetic user because it will affect the editing action choices available to the synthetic user.

The *Composition Accuracy Model* and *Editing Model* that we propose have no closely related equivalents in the research literature. It would have been quite easy to simply forego these aspects, and this would have made the design of the HGP and development of a proof-of-concept much simpler. We believe however, that both of these aspects of user HID behaviour can be modelled by attackers, and we think that this modeling is more likely in the case of the sophisticated attackers towards which this research is directed. We have therefore proposed candidate models for inclusion in the HGP.

We feel that our models are reasonable approximations of what can be modelled by attackers, but we cannot offer any evidence to prove their validity empirically. It is reasonable to ask what would be the impact of having erred in the definition of the *User Personality Model*; what would happen to this research's contribution if attackers use different models of HID user behaviour.

The major contribution of the research is the development of the HGP framework, in a manner that is consistent with a human user. We have had to develop the framework with uncertain models of HID user behaviour. It was deemed important therefore, to ensure that the HGP is conceived in a way that will allow for the integration of different models of user activity, and we argue that we have done so. The pipes and filter architecture chosen for the HGP minimizes the effect of changes to the *User Personality Model* to specific stages of the framework. Changes to a component of the *User Personality Model* requires only that the lexicon for that stage be updated and that the appropriate transformations be modified to make use of the new model. We suggest that this type of localized changes confirms the validity of the chosen framework design.

5.4.3. Feasibility

Finally, we demonstrated that the HGP that we propose here is feasible because we have been able to implement a proof-of-concept Synthetic User Environment or SUE. The SUE accepts a free-text English language `TargetDocument` and, through successive automatic transformations, produces a series of HID events which render the composition of said `Document` when placed on the USB of a compromised system. The rendering is consistent, from the perspective of the attackers' presence on the compromised system, with what would be composed by a human user including errors and their corrections.

The SUE implements the HID events associated with the use of the keyboard, but not of the mouse. We argue that the implementation of keyboard aspects is sufficient to demonstrate the feasibility of the HGP because every editing action that can be done with a mouse can be done with a keyboard. The implementation of the mouse aspects of HID behaviour is expected to be time consuming, but we do not believe that they contribute to the demonstration of validity of the HGP.

By integrating aspects of the *Editing Model* and *HID Timing Model* into the SUE, we have demonstrated that the HGP can be integrated with the *User Personality Model*. We have also demonstrated that the SUE is able to automate the transformations of the HGP and can therefore argue that the SUE is sufficient to demonstrate the validity of the HID event generation framework.

We have presented an HGP that allows for the automatic generation of mouse and keyboard activity on the USB in a manner which, from the perspective of the attackers on the compromised system, is consistent with a human user. The attackers that we have described in the *Threat* Model have complete access to the compromised system, but they are also limited by their desire to remain undetected; this affects their ability to characterise the user activity on the compromised system. We argue that our proposed HGP framework therefore, produces HID events that are consistent with a human HID user. By building a proof-of-concept SUE, we have demonstrated that the HGP is feasible and that it is a valid solution to the problem addressed by our research aim.

5.5. Contributions

This research makes a number of contributions and we list them below.

5.5.1. HGP Design

Our most important contribution is the design of a systematic approach to the generation of HID events, specifically a mouse and keyboard, in a manner that is consistent with a human inputting text into a computer system. Specifically, we have developed an *HID Generation Process* (HGP) composed of:

1. A seven stage HGP:
    a. *Stage 1 – Syntactic Elements Extraction* parses a `TargetDocument`, extracts its syntactic elements and stores them in a DPM representing their structure.
    b. *Stage 2 – Composition Errors Introduction* inserts errors in the DPM. The stage conducts three transformations on the DPM: 1) flagging nodes for errors, 2) inserting `CorrectionPoints` where the synthetic user realises that errors have been made and corrects them, and 3) inserts markers (or `Stubs`) where misplaced syntactic elements will be inserted in the `Document`.
    c. *Stage 3 – Editing Action Selection* considers the synthetic user's choices in terms of editing actions that do not depend on the `Document` or word processor. The stage has two transformations: 1) the instantiation of terminal syntactic elements which are typed by the syntactic user, and 2) the instantiation of the editing action that replaces a group of syntactic elements by another (as is the case when a wrong `Word` was used for example).
    d. *Stage 4 – HID Action Selection* takes the `Document` and word processor into account to choose the specific editing actions required to compose the `TargetDocument` including errors and their correction.
    e. *Stage 5 – Mouse Behaviour Processing* takes into effect the use of the mouse by the syntactic user, including the specification of movements and errors in the use of the mouse buttons and the missing of target locations on the screen. The stage has two main transformations: 1) the specification of mouse errors and their correction, and 2) the computation of mouse movement details.
    f. *Stage 6 – HID Event Stream Generation* generates the `KORs` and `MORs` associated with the terminal DPM nodes.
    g. *Stage 7 – Event Timing Characteristics* introduces delays between HID events and places them on the USB of the compromised system thereby rendering the composition of the `TargetDocument`.
2. *Lexicons* which represent the grammars understood by the successive stages:
    a. *i – Syntactic Element Lexicons* defines the syntactic elements which the HGP can parse such as: character, digit, document, letter, new line, number, paragraph, punctuation, signature, sentence, separators and words.
    b. *ii – Composition Error Lexicon* defines the composition errors on syntactic elements understood by the HGP, namely: misplaced,

mistyped and wrong along with correction points which may be the end of words, sentences, paragraph or document.

   c. *iii – General Editing Action Lexicon* defines general editing actions on syntactic elements, namely: move, replace and type.
   d. *iv – Specific Editing Action Lexicon* refines or defines the definitions of the following editing actions on syntactic elements: copy, cut, delete, move, paste and select. It also defines editing actions involving the mouse such as: move, use buttons and position cursor.
   e. *v – Mouse Behaviour Lexicon* defines mouse movements and errors such the use of the wrong button, the accidental pressing of a button and the missing of a target screen location.
   f. *vi – HDI Event Lexicon* defines HID `MORs` and `KORs`.
   g. *vii – Timed Event Lexicon* annotates delays between HID events.

3. A *Document Model* (`DocMod`) which represents the dynamic evolution of the `Target_Docment` as it is being composed by the synthetic user, and
4. A *Word Processor Model* (`WordProcMod`) which captures the characteristics of the application on which the synthetic user is composing the `TargetDocument`.

We have demonstrated that the architecture that we have chosen is sound and feasible by implementing it in a proof-of-concept *Synthetic User Environment* (SUE) which parses a `TargetDocument` and automatically generates the `MORs` that render its composition, incuding errors and their corrections, by placing them on the USB of the compromised system.

5.5.2. User Personality Model Definition

This research has defined the *User Personality* Model that attackers may use to characterise user activity at the USB HID level on a compromised system. As discussed further in the previous section, we have defined:

1. *Composition Model*,
2. *Typing Accuracy Model*,
3. *Editing Model*,
4. *Mouse Model*, and
5. *HID Timing Model*.

Such modelling allows network defenders to consider a further means by which attackers can characterise the environment in which they operate. The definition of these models contributes to the field of Vitality Detection, and can lead to other avenues of research.

5.5.3. NCSO and Deception Operations

This work also contributes to the evolving development of *Network Counter-Surveillance Operations*. The provision of a realistic environment on the compromised system, with which the attackers can interact, is one of three required tools of the *Network Intelligence Surveillance Toolset* that this new kind of operation requires. The framework for the automatic generation of HID events described here is an important aspect of such a realistic environment.

The ability to make it seem as if the synthetic user is composing arbitrary text on the compromised system is an important enabler to *Deception Operations*. As with the research published by Rowe [28], the HGP can be used in *lies*, *displays* and *insights*. We believe that an HGP can push *Deception Operations* further by enabling *ruses*, and *False and Planted Information*. *Ruses* involve the use of tricks such as displays that use enemy equipment and procedures; the HGP can use the compromised system, considered an asset by the attackers, to show them what defenders want them to see. The compromised system can also become a vector with which to introduce *false and planted information* to the attackers in a manner that appears much more realistic than simply introducing files in the compromised system. The use of *ruses* and *false and planted information* can help defenders prepare attackers for future operations

5.5.4. Publications

We have already contributed to the field of research through publication in [5] [4] [6]. These works argue that a reactive-oriented network defence policy based solely on perimeter defences is not sufficient to properly safeguard information technology infrastructure. An argument is made for an approach based on the idea that defence begins with an understanding of those adversaries that pose significant risk to the cyber infrastructure, their motivations and their capabilities. Therefore, the first response to an attack should not always be to immediately block the attack. The papers discuss NCSO with the objective to discover: who is attacking, what they are capable of, what their current mission objective is, and what is the larger strategic goal or context for the current attack.

5.6. Future Work

5.6.1. User Personality Models

The first aspect of future work arising from this research is in the area of user characterisation by the attackers. The likelihood that the user activity generated by the framework will be accepted as consistent with a human user by the attackers increases as our models approach those used by the attackers. Developing better models is therefore a worthwhile endeavor. Particular attention should be paid to the *Composition Accuracy Model* and *Editing Model* because there are no readily apparent research efforts detailing this type of modeling activity.

This research has devoted significant effort to supporting of the local semantics of the HID event stream. While we believe this to be important as it is a likely avenue of user activity characterisation by attackers, this is also an area that could be further explored.

5.6.2. Remaining Implementation of the HGP

The continued implementation of the automatic HGP framework is also worth further investigation. While we argue that this proof-of-concept implementation using only keyboard output reports is sufficient to demonstrate the validity of the framework, further development would serve to highlight the contributions of the work to operational intelligence collection. We suspect that the implementation of mouse movement in particular would allow for a variety of word processor models, which would increase the areas in which the synthetic user environment could be used. The implementation of mouse movements would also help refine the models of the document being composed by the synthetic user, which has the potential to make the framework applicable to a greater range of *Deception Operations*. A more robust implementation could in turn help refine the framework, further demonstrating the benefits that could be derived from it.

This research tackled the provision of a realistic user environment with which attackers can interact in order to keep them engaged with a compromised system with a view of collecting intelligence on them. We have demonstrated that this problem is worthy of significant research effort. The research proposed a framework for the automatic generation of keyboard and mouse HID events, in a manner that is consistent with a human user, and we have successfully argued for the validity of this solution. The framework was validated through the implementation of a proof-of-concept synthetic user environment, which demonstrated that the proposed solution is feasible.

REFERENCES

[1] S.P. Leblanc, "Toward The Creation Of A Synthetic User Environment - An Active Network Defence Enabler," Electrical and Computer Engineering Department, Royal Military College of Canada, Kingston, Ontario, Depth Research and Doctoral Research Proposal 2008.

[2] Computer Emergency Readiness Team (CERT). (2000, April) The CERT Division | SEI | CMR. [Online]. http://www.cert.org/tech_tips/win-UNIX-system_compromise.html#D.2

[3] Lance Spitzner, *Honeypots: Tracking Hacker*. Boston, MA: Addison-Wesley, 2003.

[4] Scott Knight, Pat Smith, Sylvain Leblanc, and David Vessey, "Living with the Enemy: Containing a Network Attacker When You Cannot Afford to Eliminate Him," in *Information Systems Technology Symposium on Information Assurance and Cyber Defence, NATO Research and TEchnology Agency*, Antalya, Turkey, 2010, pp. 25–1 – 25–10.

[5] Scott Knight and Sylvain Leblanc, "When Not to Pull the Plug: The Need for Counter-Surveillance Operations," in *The Virtual Battlefield: Perspectives on Cyber Warfare*, Christian Czosseck and Kenneth Geers, Eds. Tallinn, Estonia: Ios Press, 2009, vol. 3, ch. 16, pp. 226–237.

[6] Sylvain P Leblanc and G Scott Knight, "Engaging the Adversary as a Viable Response to Network Intrusion," in *Workshop on Cyber Infrastructure - Emergency Preapardness Aspects*, Ottawa, Canada, 2005.

[7] R.W. Smith and G.S. Knight, "Predictable Design of Network-Based Cover Communications," in *IEEE Symposium on Security and Privacy (S&P 2008)*, Oakland, CA, 2008, pp. 311–321.

[8] Yingxu Wang, "A formal syntax of natural languages and the deductive grammar," *Fundamenta Informaticae*, vol. 90, no. 4, pp. 353–368, April 2009.

[9] D.J. Major, "Exploiting system call interfaces to observe attackers in virtual machines," Royal Military College of Canada, Kingston, Ontario, MASc Thesis

2008.

[10] Harley B. Heywood, "An Intrusion Surveillance Toolset Based on Virtual Machine Introspection," Royal Military College of Canada, Kingston, Ontario, MASc Thesis 2011.

[11] T.Dean, G.S. Knight J.S. Alexander, "Spy vs. Spy: Counter-Intelligence Methods for Backtracking Malicious Intrusions," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'11)*, Markham, Ontario, 7-10 November 2011, pp. 1–14.

[12] Cliff Stoll, *The cuckoos egg: tracking a spy through the maze of computer espionage*. New York, NY: Simon & Schuster, 2005.

[13] Bill Cheswick, "An Evening with Berferd in which a cracker is Lured, Endured, and Studied," in *Winter USENIX Conference*, San Francisco, 1992, pp. 103–116.

[14] Neal Krawetz, "Anti-honeypot technology," *Security & Privacy, IEEE*, vol. 2, no. 1, pp. 76–79, Jan-Feb 2004.

[15] Kyumin Lee, James Caverlee, and Steve Webb, "Uncovering social spammers: social honeypots+ machine learning," in *33rd International ACM SIGIR conference on Research and development in information retrieval*, New York, NY, 2010, pp. 435–442.

[16] Chris Brenton, "Honeynets," in *SPIE Enabling Technologies for Law Enforcment and Security*, Boston, MA, 2001, pp. 115–122.

[17] John Levine, Richard LaBella, Henry Owen, Didier Contis, and Brian Culver, "The use of honeynets to detect exploited systems across large enterprise networks," in *Information Assurance Workshop*, West Point, NY, June 2003, pp. 92–99.

[18] Abdallah Ghourabi, Tarek Abbes, and Adel Bouhoula, "Characterization of attacks collected from the deployment of Web service honeypot," *Security and Communication Networks*, vol. 7, no. 2, pp. 338–351, February 2014.

[19] Niels Provos and Thorsten Holz, *Virtual honeypots: from botnet tracking to intrusion detection*, 1st ed. Boston, MA, USA: Pearson Education Professional, 2007.

[20] Zubair A Khan, Saeed U Rehman, and M H Islam, "An analytical survey of state of the art wormhole detection and prevention techniques," *International Journal of Science and Engineering REsearch*, vol. 4, no. 6, pp. 1723–1731, June 2013.

[21] Checkpoint Security. (2014, February) Threat Prevention Appliances | Checkpoint Software. [Online]. http://www.checkpoint.com/products/threat-prevention-appliances/

[22] Communications Security Establishment Canada (CSEC). (2014, February) Baseline Security Requirements for Network Security Zones in the Government of Canada. [Online]. http://www.cse-cst.gc.ca/its-sti/publications/itsg-csti/itsg22-eng.html

[23] Miles A McQueen and Wayne F Boyer, "Deception used for cyber defense of control systems," in *2nd conference on Human System Interactions HSI*, Catania, Italy, May 2009, pp. 624–631.

[24] Dan Ragsdale, "Scalable Cyber Deception," Defence Advanced Research Project Agency (DARPA), Arlington, VA, Briefing Charts 2011.

[25] James B Michael, "On the response policy of software decoys: Conducting software-based deception in the cyber battlespace," in *Computer Software and Applications Conference COMPSAC*, Oxford, UK, 2002, pp. 957–962.

[26] Jim Yuill, Fred Feer, Dorothy Denning, and Bowyer Bell, "Deception for Computer Security Defense," Office of the Secretary of Defense - The Pentagon, Washington, DC, research project final-report for the 2004.

[27] Neil C Rowe, "A model of deception during cyber-attacks on information systems," in *Multi-Agent Security and Survivability*, August 2004, pp. 21–30.

[28] Neil C Rowe, "Counterplanning deceptions to foil cyber-attack plans," in *Information Assurance Workshop*, West Point, NY, June 2003, pp. 203–210.

[29] James F Dunnigan and Albert A Nofi, *Victory and Deceit: Deception and Trickery at War*. Lincoln, NE, USA: Writers Club Press, 2001.

[30] Neil C Rowe and Hy S Rothstein, "Two taxonomies of deception for attacks on information systems," *Journal of Information Warfare*, vol. 3, no. 2, pp. 27–39, July 2004.

[31] Osama Hayatle, Amr Youssef, and Hadi Otrok, "Dempster-Shafer Evidence Combining for (Anti)-Honeypot Technologies," *Information Security Journal: A Global Perspective*, vol. 21, no. 6, pp. 306–316, December 2012.

[32] Bill McCarty, "The honeynet arms race," *Security & Privacy, IEEE*, vol. 1, no. 6, pp. 79–82, Nov-Dec 2003.

[33] Joseph Corey. (2003, September) Phrake Fakes. [Online]. URL http://www. phrack. org/fakes/p62/p62-0x07. txt, http://www. phrack. org/fakes/p62/p62-0x07. txt

[34] John Clark, Sylvain Leblanc, and Scott Knight, "Compromise through USB-based Hardware Trojan Horse Device," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 555–563, May 2011.

[35] Yogendra N Singh and Sanjay K Singh, "A taxonomy of biometric system vulnerabilities and defences," *International Journal of Biometrics*, vol. 5, no. 2, pp. 137–159, April 2013.

[36] Issa Traore and Ahmed A Ahmed, *Continuous authentication using biometrics: Data, models, and metrics*. Hershey, PA, USA: IGI Global, September 2012.

[37] Zach Jorgensen and Ting Yu, "On mouse dynamics as a behavioral biometric for authentication," in *6th ACM Symposium on Information, computer and Communications Security*, Hong Kong, March 2011, pp. 476–482.

[38] Ahmed A Ahmed and Issa Traore, "Detecting Computer Intrusions Using Behavioral Biometrics.," University of Victoria, Victoria, BC, Technical Report 2005.

[39] Ahmed A Ahmed and Issa Traore, "A new biometric technology based on mouse dynamics," *Dependable and Secure Computing, IEEE Transactions on*, vol. 4, no. 3, pp. 165–179, July-Sept. 2007.

[40] Maja Pusara and Carla E Brodley, "User re-authentication via mouse movements," in *workshop on visualization and data mining for computer security*, Washington, DC, 2004, pp. 1–8.

[41] Salil P Banerjee and Damon L Woodard, "Biometric authentication and identification using keystroke dynamics: A survey," *Journal of Pattern Recognition Research*, vol. 7, no. 1, pp. 116–139, 2012.

[42] Daniele Gunetti and Claudia Picardi, "Keystroke analysis of free text," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, pp. 312–347, 2005.

[43] Francesco Bergadano, Daniele Gunetti, and Claudia Picardi, "Identity verification through dynamic keystroke analysis," *Intelligent Data Analysis*, vol. 7, no. 5, pp. 469–496, January 2003.

[44] Livia C Araujo, Luiz H Sucupira Jr, and et al., "User authentication through typing biometrics features," *Signal Processing, IEEE Transactions on*, vol. 53, no. 2, pp. 851–855, Feb 2005.

[45] Alen Peacock, Xian Ke, and Matthew Wilkerson, "Typing patterns: A key to user identification," *Security & Privacy, IEEE*, vol. 2, no. 5, pp. 40–47, Sept-Oct 2004.

[46] Fadhli W Wong, Ainil S Supian, Ahmad F Ismail, Lai W Kin, and Ong C Soon, "Enhanced user authentication through typing biometrics with artificial neural networks and k-nearest neighbor algorithm," in *35th Asilomar Conference on Signals, Systems and Computers*, vol. 2, Pacific Grove, CA, Noveember 2001, pp. 911–915.

[47] Willem G De Ru and Jan H Eloff, "Enhanced password authentication through fuzzy logic," *IEEE Expert*, vol. 12, no. 6, pp. 38–45, Nov-Dec 1997.

[48] Wasil E Eltahir, MJE Salami, Ahmad F Ismail, and WK Lai, "Dynamic keystroke analysis using AR model," in *Intl. conference on Industrial Technology*, vol. 3,

Hammamet, Tunisia, December 2004, pp. 1555–1560.

[49] Bonnie E John and David E Kieras, "The GOMS family of user interface analysis techniques: Comparison and contrast," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 3, no. 4, pp. 320–351, December 1996.

[50] I Scott MacKenzie and William Buxton, "Prediction of pointing and dragging times in graphical user interfaces," *Interacting with Computers*, vol. 6, no. 2, pp. 213–227, June 1994.

[51] USB.org. (2014, February) USB.org - USB 2.0 Documents. [Online]. http://www.usb.org/developers/docs/usb20_docs/

[52] Ken Shoemake, "Euler angle conversion," in *Graphics Gems*, Paul Heckbert, Ed. San Diego, CA, USA: AP Professional, 1994, ch. III.5, pp. 222–229.

[53] Géry Casiez and Nicolas Roussel, "No more bricolage!: methods and tools to characterize, replicate and compare pointing transfer functions," in *24th Annual ACM symposium on User interface softare and technology*, Honolulu, Hi, 2011, pp. 603–614.

[54] PLX Technologies. (2013, Dec.) PLX Technology : Legacy USB Controller. [Online]. http://www.plxtech.com/products/usbcontrollers/legacy#net2280

[55] Don Ho. (2013, Dec.) Notepad++ Home. [Online]. http://notepad-plus-plus.org/

[56] Microsoft. (2013, Dec.) Windows XP Home Page. [Online]. http://www.microsoft.com/canada/windowsxp/default.mspx

[57] Microsoft. (2013, Dec.) Windows - Microsoft Windows Help. [Online]. http://windows.microsoft.com/en-ca/windows/windows-help#windows=windows-7

[58] Eclipse Foundation. (2013, Dec.) Eclipse Downloads. [Online]. http://www.eclipse.org/downloads/

[59] Queen's University at Kingston. TXL Documentation. [Online]. http://txl.ca/ndocs.html

[60] James R Cordy, Ian H Carmichael, and Russell Haliday. (2012, July) TXL Documentation. [Online]. http://www.txl.ca/docs/TXL106ProgLang.pdf

[61] Thomas R Dean, James R Cordy, Andrew J Malton, and Kevin A Schneider, "Grammar programming in TXL," in *2nd IEEE International workshop on Source Code Analysis and Manipulation*, Montréal, QC, October 2002, pp. 93–102.

[62] ISO, "IEC 14977: 1996 (e), Information Technology Syntactic Metalanguage Extended BNF," International Standard 1996.

[63] John T Clark, "On Unintended USB Communication Channels," Royal Miliary College of Canada, Kingston, MASc Thesis 2009.

APPENDICES

Appendix A : I – SYNTACTIC ELEMENTS LEXICON

The Syntactic Elements Lexicon(SynElmtLex) describes the various syntactic elements that are recognized by the HGP. It is used by the first stage of the SUE Event Generation Process, Syntactic Element Extraction.

The SynElmtLex contains the following Syntactic Element:

```
Character
    Digit | <EOT> | Letter | New_Line  | Punctuation |
  Separator


Digit
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }


Document
    1..* Paragraph, [Signature], <EOT>


Letter
    { A, …, Z, a, …, z }


New_Line
    ( <Carriage Return>, <Line Feed>) | <Line Feed>


Number
    [Separator] [-] [$] 1..* Digit [ ., 1..* Digit]
  [%][Separator]


Paragraph
    1..* Sentence, New_Line


Punctuation
    { \,, .,  ;, !, ?, : }


Signature
    [0..* Words], New_Line
```

```
Sentence
    0..* SentenceElement , Punctuation

SentenceElement
    Word  | Number
```

```
Separator
    { \\, \<,\ >, \(, \), \{,\} , \" }

Word
    [Separator] 1..* Letter [Separator]
```

Notation:

`,`	Enumeration or list separator
`|`	OR representing a choice between elements.
`? .. ?`	Sequence of specified cardinality, where cardinality can be 0, 1 or many (*)
`{ }`	Comma separated enumeration
`[ ]`	Comma separated list of optional elements
`( )`	Comma separated list of mandatory elements
`< >`	Non-printable character
`\`	Escape character for notation tokens: ii- Composition Error Lexicon

Appendix B : II – COMPOSITION ERROR LEXICON

The CompErrorLex describes the Composition Errors that are implemented by the HGP. The CompErrorLex is based on the characteristics of the User Personality Models, and is used by Stage 2 of the HGP.

```
CompError
    Mispalced_SE |   Mistyped_SE |  Wrong_SE
```

`Misplaced_SE` A misplaced Syntactic Element is out of position in the current context, compared to its correct position in the `TargetDocument`. The error will be implemented by inserting a new node (Stub) in the DPM to duplicate the Target out of Position.

1. Error ID of the form `Misplaced_SE_Unique#`
2. Position of the form `{Direction, Magnitude}`
   a. Direction indicates whether the Syntactic Element is misplaced early (`Left` in the `Document`) or late (`Right` in the `Document`)
   b. Magnitude indicates by how much the Syntactic Element is misplaced, expressed in `#paragraphs ; #sentences ; #sentence_elements`. The current implementation only allows for SyntacticElements to be misplaed by level Paragraph by Paragraph¸Sentence by Sentence, etc. The current implementation only allows Words to be misplaced.
3. `Move_SE  {Position} | Delete_SE, Type_SE`
4. `DPM_Node > Erroneous_DPM_Node`
5. `StubInserted`

`Mistyped_SE` Erroneous sequence of `Character` (sequence not representing the Target Syntactic Element, where either: 1) additional `Characters` are introduced, or 2) legitimate `Characters` are typed out of sequence

1. Error ID of the form `Mistyped_SE_Unique#`
2. `{ 0..* Number, 0..* Punctuation, 0..* Separator, 0..*Word}`

3. `Replace_SE`
4. `DPM_Node > Erroneous_DPM_Node`
5. `CorrectionPointInserted`

`Wrong_SE` A wrong Syntactic Element is typed instead of the intended Syntactic Element contained in the `TargetDocument`.

1. Error ID of the form `Wrong_SE_Unique#`
2. `{Word | Sentence | Paragraph}`
3. `Replace_SE`
4. `DPM_Node > Erroneous_DPM_Node`
5. `CorrectionPoint Inserted`

Appendix C : III – GENERAL EDITING ACTION LEXICON

The GenEditActionLex describes the various `Document` editing actions that are processed by the HGP. General editing actions are those chosen by the user without regard to the structure of the Document or Word Processor.

The GenEditActionLex introduces the following Editing Actions, none of which depend on the Document or Word Processor structure:

```
Move_SE          Change the position of a Syntactic Element in the Document
    {Position}

Replace_SE       Replaces a Syntactic Element with another in the Document
    Delete_SE, Type_SE
  | Select_SE, Paste_SE
  | Select_SE, Delete_SE, Paste_SE

Type_SE          Input a Syntactic Element in the Document at the Cursor
                 location, one character at a time using the keyboard.
    1..* Character
```

The DocEditActionLex also redefines the following syntactic elements to show them instantiated with a Type_SE node

```
New_Line
    ( <Carriage Return>, <Line Feed>)

  | <Line Feed> [Type_SE]


Number
    [Separator][-] [$] 1..* Digit [ ., 1..* Digit]
  [%][Separator]     [Type_SE]

Punctuation
    { \,, .,  ;, !, ?, : } [Type_SE]


Separator
    { \\, \<,\ >, \(, \), \{,\} , \" } [Type_SE]
```

```
Word
    [Separator] 1..* Letter [Separator] [Type_SE]
```

Mistyped_SE

1. Error ID of the form Mistyped_SE_Unique#
2. { 0..* Number, 0..* Punctuation, 0..* Separator, 0..*Word} [Type_SE]
3. Replace_SE
4. DPM_Node > Erroneous_DPM_Node
5. CorrectionPointInserted

Wrong_SE

1. Error ID of the form Wrong_SE_Unique#
2. {Word | Sentence | Paragraph} [Type_SE]
3. Replace_SE
4. DPM_Node > Erroneous_DPM_Node
5. CorrectionPoint Inserted

Appendix D : IV – SPECIFIC EDITING ACTION LEXICON

The Specific Editing Action Lexicon (`iv-SpecEditActionLex`) contains the following Editing Actions, which are all dependent on the Document or Word Processor structure, along with means of navigation (through Mouse and Keyboard) and means of interacting with the Word Processor (through the Mouse):

`Copy_SE`	Place Syntactic Element on the application clipboard. This command corresponds to the `Copy` action of the `Edit_Menu`.

```
    Menu_Area.Edit_Menu.Copy |<Ctrl-C> | RC.Edit_Menu.Copy
```

`Cut_SE`	Place Syntactic Element on the application clipboard and remove it from the Document. It corresponds to the `Cut` action of the `Edit_Menu`.

```
  Menu_Area.Edit_Menu.Cut | <Ctrl-X> | RC.Edit_Menu.Cut
```

`DC`	Double-click left mouse button.
`Delete_SE`	Removes a Syntactic Element from the Document.

```
    Select__SE, <DEL>
  | Select__SE, <Backspace>
  | Position_Cursor, 1..* <Backspace>
  | Position_Cursor, 1..* <DEL>
```

`LC`	Click left mouse button
`LR`	Release left mouse button
`Move_Mouse`	Move the `Pointer` from one location to another in the Document.

```
    [Up | Down] #Rows, [Left | Right] #Colums
  | Menu_Name.Menu.Element
```

`Move_SE`	Move a Syntactic Element from one location to another in the Document.

```
    Select_SE, Cut_SE, Position_Cursor, Paste_SE
  | Select_SE, LC, Drag, LR
```

`Nav_Arrows`

    Left_Arrow (LA), DA, RA, UA

Paste_SE	Place a Syntactic Element from the application clipboard to the Document, in the position corresponding to the Cursor's location.

    Menu_Area.Edit_Menu.Paste | <Ctrl-V> | RC.Edit_Menu.Cut

Position_Cursor	Position the Cursor to a specific Document location in the application's Editing_Area.

    Move_Mouse, LC, LR | 1..* Nav_Arrows

RC	Click right mouse button
RR	Release right mouse button
Select_SE	Highlight a Syntactic Element in the Document.

```
    Move_Mouse, DC
  | Move_Mouse, LC, Drag, LR
  | Position_Cursor , <Shift>, 1..* Nav_Arrows>
```

Appendix E : V – MOUSE BEHAVIOUR LEXICON

The Mouse Behaviour Lexicon (v-MouseBehaviourLex) defines Move_Mouse and MouseErrors. Each MouseError captures four elements:

1. The Target - the intended mouse action.
2. The Error – the erroneous mouse actions.
3. The Rectifying Actions – the sequence of Editing Actions required in order to transform the Erroneous Syntactic Element into the Target Syntactic Element.
4. The Correction Point – in terms of the Target Syntactic Element's context, represented as the DPM node after which the rectifying actions will be carried out.

v-MouseBehaviourLex contains the following elements:

Accident_But	A mouse button is inadvertently pressed.

```
   1. NULL
   2. RC, RR | LC, LR
   3. <ESC> | Position_Cursor
```

Missed_Loc	Mouse_Indicator does not hit the desired location during MM.

```
   1. Move_Mouse
   2. Move_Mouse
   3. Move_Mouse
```

Move_Mouse	Mouve the Pointer to a specific Screen_Coordinate at a Speed expressed in pixel/s.

```
   (X,Y), Speed
```

Wrong_But	Wrong Button when the wrong mouse button is pressed.

```
   1. LC, LR |  RC, RR
   2. RC, RR | LC, LR
   3. <ESC>, 0..* MM, LC, LR | RC, RR
   4. Erroneous_DPM_Node + 1
```

Appendix F : VI – HID EVENT LEXICON

The HIDEventLex contains the following HID Events:

KOR The Keyboard Output Report as specified by the USB HID Class with Timing Information

Offset	7	6	5	4	3	2	1	0	Remark
0	RGUI	RAlt	RShift	RCtl	LGUI	LAlt	LShift	LCtl	Modified Keys
1	Reserved								Ignored
2	Key Code 1								Key Arrays
…	…								
7	Key Code 6								

MOR The Mouse Output Report as specified by the USB HID Class with timing information

Offset	7	6	5	4	3	2	1	0	Remark
0	Device Specified					MidBut	RightBut	LeftBut	User Input
1	X Displacement								
2	Y Displacement								

Appendix G : VII – TIMED EVENT LEXICON

The Timed Event Lexicon (`vii-TimedEventLexicon`) introduces a release time to the HID Events defined previously. It contains the following elements:

`tn`	Release time, expressed in milliseconds, since the beginning of emissions of the HID event stream on the USB
(KOR, t_n)	Keyboard Output Report followed by its release time.
(MOR, t_n)	Mouse Output Report followed by its release time.