UNIVERSITY OF CALGARY

Evaluating the Emergent Effects of (Multiple) Security Mechanisms via Evolutionary Algorithms

by

Jonathan William Hudson

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER, 2018

Abstract

Security mechanisms provide protection against system penetration and exploitation by providing coverage for vulnerabilities. However, security mechanisms often have demanding operational requirements that necessitate access to system resources and control of monitoring points. At the same time, users have particular requirements from programs they install, how they interact with these programs, and what performance they expect from their computing system. These combined requirements create a selection problem where the user desires to balance security coverage, through a choice of security mechanism(s), with system performance and functionality. This problem is known as the Effective Security-in-Depth problem. First, this thesis introduces a genetic algorithm to enable an evolutionary search for interaction event sequences for the problem of Effective Security-in-Depth. This methodology required the development of a fitness function that integrated numerous system metrics while addressing the variance found in event sequence simulation and measurement. Next, the steps for effectively implementing this methodology as a software tool are described. Finally, this thesis introduces three processes to use the tool to select between single security mechanisms for different usage profiles, compare and contrast subsets of security mechanisms, and evaluate examples of emergent misbehaviour such as system failure. The initial experimental evaluation validates the ability of the search for interaction event sequences to make progress despite the challenges of stochastic system measurement. The remaining experimental evaluations demonstrate the success of an application of each of the three processes. The evaluation supports that the developed method, tool, and processes are a viable solution to the problem of Effective Security-in-Depth.

Acknowledgments

The support of my supervisor Dr. Jörg Denzinger, the helpful and amazing staff and faculty in the Department of Computer Science, and all the graduate student friends made along the way.

Dedication

To my parents Art and Bonita, who made this thesis possible, the understanding support of my wife Stephanie, the patience of my sister Sarah, and to the memory of my brother Daniel and sister Anna.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter 1

Introduction

Information technology security includes numerous different physical, business, and computing practices, which protect information technology systems. One of the most important information technology security practices is the choice of what security mechanism(s) to use to protect a computer system. There are numerous choices of which security mechanism(s) to use, since each provides a different range of prevention and/or detection of unauthorized actions by users and programs. Hardware and software technologies are continuously developing, replaced, and inter-mixed for heterogeneous operation. At the same time, each user and/or organization will have different operational requirements and skill-sets that create demands on the technology assets' performance. The balance of these performance requirements and security is a key challenge for procurement and deployment of information technology assets.

This thesis is concerned with improving the effective management of computer security mechanisms through the formal definition of the problem and the development of an exploratory search methodology. In particular, this effective management is concerned with two challenges. The first challenge is that of an individual user selecting a mechanism that provides a desired level of security coverage while minimizing the performance consequences in respect to the user's expected usage profile. The second challenge is concerned with the additional inter-operational performance challenges created when using multiple security mechanisms simultaneously. The methodology developed in this thesis is accompanied by the creation of the necessary tools and processes to evaluate, compare, and select between security mechanisms and their combinations. This selection goal is to achieve effective security protection given a

particular expected user usage profile.

The preferences of the individual users of a computer system are often the least explored entity involved in information technology security. Different users will have divergent usage profiles consisting of what programs they install and what interactions with those programs they utilize. These individual usage profiles consequently result in users having different performance demands. The performance challenges resulting from a combination of security mechanisms are generally different based on the user's usage profile. One user may find the performance demands of a particular security mechanism disruptive based on their expected usage, while another user may not. Identifying which security mechanism(s) to choose is a challenging problem given the variety of program interactions and the possible orders in which they may occur.

It is valuable for the manager of an IT system to secure their system by installing security mechanisms. The resultant prevention and deterrence of attacks typically provides a better return on investment than recovering after successful attacks [Ros08]. However, the pursuit of security necessarily must balance the other operational requirements of the secured system. For example, the design of a particular security mechanisms may conflict with programs the users are required to use daily. In another example, although the ideal secured system would consist of a multitude of security mechanisms for each area of coverage, too many security mechanisms will likely result in competition for operating resources and control of monitoring points [LBS09]. The usage of operating resources by security mechanisms reduces the resources available to the rest of the computing system. Additionally, the execution of code to monitor critical points in the system requires computing cycles to be diverted and delays every action that triggers monitoring.

Security mechanisms provide coverage for vulnerabilities where exploitation will result in unwanted behaviour within the system [Gol11]. The use of multiple security

mechanisms to cover a type of vulnerability is known as Security-in-Depth, which is the technology part of Defence-in-Depth. Defence-in-Depth is the broader strategy for information technology security that includes the technology, people, and operational methods used to provide layered security protection [Nat12]. Security-in-Depth is the layering or simultaneous use of more than one security mechanism within a computing system [Smi03]. The goal of layered defence is that bypassing one layer of security or mechanism protecting a vulnerability still leaves further layers to challenge the attacker [Sua03]. Layered defence is designed to provide redundancy, to the coverage of the secured system, and to increase the investment necessary for the attacker to achieve success.

Security mechanism designers will often have different ideas on how to achieve security coverage. These ideas will generally have different strengths and weaknesses. Ideally the combination of security mechanisms would co-operate to make use of all their strengths while covering for each of their weaknesses [DFF97]. However, there are challenges to trusting security mechanisms to work together cooperatively [LBS09, Ros08]. Research and software development goals around security are often differently focused. It is usually in a developer's interest to either ignore inter-operation or to make their mechanism incompatible with competitors [Per11, Sky06]. In other cases, security researchers are often focused on creating solutions in their specific research field resulting in highly targeted solutions that often de-emphasize the greater scope of performance or interoperability. In addition, there are confirmations of state actors compromising security mechanisms via targeted influence [Coh13]. This lack of trust results in complications for naive security mechanism collaboration.

This thesis is concerned with the achievable goal of Effective Security and furthermore Effective Security-in-Depth. The Effective Security problem is the determination of a security mechanism that maximizes the system's functionality and per-

formance while maintaining, or exceeding, a desired level of security coverage. The Effective Security-in-Depth problem extends this problem to consider combinations of security mechanisms. Consequently, the Effective Security problem is in effect the Effective Security-in-Depth problem for subsets of security mechanisms of size one.

This thesis addresses the pessimistic formulation of the problem of achieving Effective Security-in-Depth in which security mechanisms are presumed to have undesirable performance consequences and to have conflicts in inter-operation. This thesis develops a methodology to evaluate the performance of security mechanism combinations, an implementation tool to enable the application of the methodology, and processes to use this tool. Particularly, this thesis evaluates combinations of security mechanisms, which provide the desired level of security coverage, to determine if they have unacceptable functional problems and/or comparative performance handicaps.

Ordinarily, considerations of security deployment are done operationally by system administrators. In a typical business environment, this often results in trusting a single vendor to provide a security suite, or to rely on a single vendor's mechanism for each coverage area [Per11]. Both solutions commonly result in a single layer of coverage. In fact, many security mechanisms, or their creators, actively advise against simultaneous deployment and operation of other vendor's software alongside their own. Security-in-Depth is designed to take advantage of the strengths of multiple developers' security mechanisms by using more than one for a coverage area. The evaluation of these more complex security suites is challenging since it requires access to, and time to comparatively test, several differing security mechanism combinations.

Emergent behaviour is a further challenge to the problem of determining Effective Security-in-Depth. Emergent behaviour is unpredictable behaviour that emerges from the interaction of multiple entities such as a collection of security mechanisms [Mog06]. Emergent behaviour is unpredictable from the examination of each security

mechanism operating independently. Similarly, given that collaboration is not a common design goal, then inter-operation is most likely to result in negative functional or performance consequences. Such emergent behaviour can only be discovered by exploring and exposing it during the security mechanisms' inter-operation [Hol98].

The formulation of the Effective Security-in-Depth problem, as a system of multiple agents that exhibit unpredictable emergent behaviour, allows the application of methods successful in multi-agent system research. It has been demonstrated in [HDKB11, HD15] that the evolution of interaction event sequences could be used to explore the extent of unpredictable emergent misbehaviour in a multi-agent system. In this thesis, this concept is re-structured to apply to the context of Effective Security-in-Depth. This re-structuring requires unique considerations to be made for the challenges of performance evaluation and event sequence simulation necessary to quantify and explore the Effective Security-in-Depth problem.

To begin, this thesis develops a formal definition of the Effective Security-in-Depth problem before introducing an evolutionary search for event sequences guided by a uniquely designed fitness function. Running a security mechanism, like any program, has performance costs. The goal of the developed fitness function is to separate the typical performance costs from the costs of encountering a specific sequence of events, or from the unique costs resulting from inter-operation with other security mechanisms. The secondary goal of the design of the fitness function and search process is to account for the stochastic nature of system operation and measurements of performance.

The developed fitness function combines a variety of system measures quantifying the performance and functionality of the secured system. Simulated event sequence performance on emulated hardware with installed security mechanisms is evaluated and compared relative to that of an unsecured system. Specifically, this compari-

son considers the chosen fitness function component metrics relative to a representative sample distribution from both the secured and unsecured systems. The fitness function identifies where system performance metrics were more negatively affected relative to the secured system distribution than they were relative to the unsecured system distribution. At the same time, to deal with the stochastic nature of the metrics, multiple evaluations of specific individuals are merged, and the median fitness is used for the evolutionary search selection process. The experimental evaluation in this thesis of an implementation of this developed methodology demonstrates that the fitness function of the proposed method is successful in directing the evolutionary search method's progress. This successful progress is made despite the challenges of dynamic simulation 'noise' in measuring a point in the problem's search space.

Next, the developed evolutionary search methodology is implemented as a tool designed to allow system operators to examine security mechanisms through a virtual machine set-up. This set-up allows for event sequences to be executed for different configurations of security mechanisms. The experimental evaluation of this implementation explores how examples discovered by the exploratory search can then be examined by the search operator. Such examples include those where system operation experiences an unacceptable slowdown in time, and instances where the system is unable to complete a sequence of interactions. These examples can be used to choose between prospective security mechanisms for different user usage profiles.

Lastly, this thesis examines the processes for utilizing the evolutionary search methodology implemented in the previous tool to verify, evaluate, and comparatively examine different configurations of security mechanisms. The exploration and evaluation of many different security suite configurations under the large space of possible interaction event sequences is an exhausting repetitive challenge. The developed tool automates this exploration allowing for human intelligence to be more productively

applied via the described processes that examine the produced results. The experimental evaluation of these processes explores using the software tool to examine performance costs, contrast and select between different security suites, and verify operational problems created by a subset of security mechanisms.

The structure of this thesis is as follows. First, Chapter 2 provides the formalized definition of the Effective Security-in-Depth problem established in this thesis. This definition is necessary to communicate the remaining developments of this thesis. Next, in Chapter 3, the research areas related to Effective Security-in-Depth and exploratory testing via event sequences are examined. Chapter 4 provides the basic definitions for genetic algorithms. These definitions are used in Chapter 5 to define the abstract methodology of evolutionary search for interaction event sequences, as well as the construction of the fitness function to direct it for the problem of Effective Security-in-Depth. Chapter 6 describes the implementation of the abstract evolutionary search mechanism as a tool. Next, processes to use the implemented tool for the Effective Security-In-Depth problem are established in Chapter 7. Chapter 8 explores empirical evaluations that demonstrate the applicability of the methodology, tool, and processes introduced in the previous three sections. Lastly, Chapter 9 concludes the work and gives an outlook on possible future work.

Chapter 2

Effective Security-in-Depth

This chapter formalizes the Effective Security-in-Depth problem and provides its technical definitions. This formalization is necessary to communicate the remaining developments described in this thesis. First, Section 2.1 situates where the Effective Security-in-Depth problem exists as a challenge, and defines the boundaries of the work done in this thesis in regard to it. This is followed, in Sections 2.2 and 2.3, by the formal description of the Security-in-Depth and Effective Security-in-Depth problems. With this description in hand, the subsequent chapter discusses related research areas and the state-of-the-art within them.

2.1 Security-in-Depth Background

Defence-in-Depth is a strategy in which Security-in-Depth, sometimes called layered defence, is only the first tactic employed [Sma11]. Defence-in-Depth is often misidentified as the more specific sub-area Security-in-Depth. Within Defence-in-Depth, Security-in-Depth provides a layered defence against which the attacker is supposed to expend their resources in an unsustainable manner. Defence-in-Depth follows this frustrating defensive structure with a planned counter-attack against an exhausted or exposed opponent [Sua03].

As a greater strategy, Defence-in-Depth is best suited for physical confrontations where the enemy risks true exposure during an attack. The digital form of Defence-in-Depth is harder to carry out. The physical distance separating parties, along with the undefined boundary between civil, criminal, and military actions, make international counter-attack and/or prosecution difficult [Sma11]. This thesis does not address

any strategic response, if one of the security mechanisms is triggered or an attacker identified. This thesis is focused on the aforementioned step of first achieving layered defence in a way that is acceptable to the user.

Security-in-Depth is a highly desirable property for a secured system to have due to the layers of security protection it creates [Smi03]. However, successful achievement of Security-in-Depth is difficult to ascertain [Sma11]. The ideal formulation of the Security-in-Depth problem relies on the assumptions that the overall exposed area of the system can be identified and that the coverage of each security mechanism for that area can be defined. On the other hand, in general it is not possible to enumerate the security vulnerabilities of a system and therefore not possible to determine the exact coverage area of a single security mechanism. Examples of types of security mechanisms include intrusion detection systems, virus scanners, firewalls, file monitors, and access control systems.

The most realistic formulation of the Security-in-Depth problem requires some assumption of what the overall exposed security coverage area is, and an acceptance of the coverage claims provided by security mechanism designers. This thesis does not attempt to verify or explore the security claims made by any utilized security mechanisms. The claimed coverage area, or in the case of modern anti-virus suites the multiple areas of claimed coverage, are accepted as stated. The verification of claimed coverage is best served by specific dedicated research, examination, and evaluation. As a result, the formulation of the Security-in-Depth problem in this thesis is the selection of a subset of security mechanisms from the complete set of available mechanisms such that the subset provides the desired level of protection in respect to the claimed coverage areas.

The simplest coverage solution consists of providing a single layer of complete coverage by selecting from security mechanisms with disjoint coverage areas. This

is generally accomplished by system operators either selecting a single vendor who claims to provide a full security coverage suite of security mechanisms, or by selecting a single mechanism per perceived coverage area.

The goal of Security-in-Depth is properly achieved when there is more than a single layer. Multiple layers are preferred since they provide depth of protection in regard to a vulnerable area [Lon11]. However, there are a number of practical realities that make Security-in-Depth challenging to achieve.

1. Security mechanisms are rarely disjoint in coverage areas.

   Many security companies attempt to provide all-around solutions to consumers in order to receive all their funding dedicated towards security [Per11].

2. Security mechanisms are rarely disjoint in operation.

   The fiscal, research, or business goals that motivate the creation of a security mechanism mean that they are often designed in a silo approach, which either unintentionally, willfully, or maliciously ignores the operation of other security mechanisms of the same or differing type [LBS09].

The challenge of Security-in-Depth is to achieve the desired coverage despite the fact that the security mechanisms may not have disjoint coverage areas or may have operational incompatibilities.

More importantly, it is possible to simply activate, install, or operate a large quantity of security mechanisms to exceed the desired layers of coverage. However, this naturally comes with the increased load and resources required by multiple security mechanisms. The consequences of these requirements and load may negatively impact the performance of the secured system. This negative impact may take the form of slower response time for disk, network, or processor access and/or an increased length of time to complete operations requiring processing cycles.

Outside of circumstances requiring security at the sacrifice of everything else, it is generally unacceptable to make a system unusable as the result of performance degradation. In many cases, perceived degradation of performance or experience generally results in the security mechanisms being disabled or circumvented by system users. For example, the users of the popular PC computer game service Steam are generally advised in forums to uninstall security mechanisms to achieve greater performance and responsiveness [Val13].

The term Shadow IT is used to describe when Information Technology (IT) systems and solutions are built or used in an organization without the awareness of the IT department [SB14]. This often occurs when users determine that the IT department's configuration of security impedes their productivity and attempt to find ways to circumvent or disable those protections.

Therefore, the realistic operational compromise that defines the Effective Security-in-Depth problem is fulfilling the desired level of coverage, while maintaining operational compatibility and acceptable system performance. This acceptable performance level is dictated by the requirements of the user. The performance of the system is measured quantitatively by metrics gathered from a system with the mechanisms installed when it is exposed to network, disk, and/or processing load.

2.2 Security-in-Depth

The definition of the Effective Security-in-Depth (ESiD) problem is assisted by first formally defining the Security-in-Depth (SiD) problem. The SiD problem consists of determining a subset of security mechanisms that achieve a desired depth of coverage, without any regard to the performance of the protected system. The Effective SiD problem additionally limits covering subsets of security mechanisms to those that achieve the desired functionality and minimize unwanted performance consequences.

ESiD problem solutions are therefore a subset of the SiD solutions. That is, all effective covering subsets of security mechanisms provide the desired coverage, but not all covering subsets fulfill the desired effective performance and functionality.

The Security-in-Depth (SiD) problem is the selection of a collection of security mechanisms that together provide a desired depth of security coverage. The definition of the SiD problem uses the following formalizations.

- C is the set of n known areas of coverage such that $C = \{c_1, c_2, \ldots, c_n\}$.

  Coverage areas can be broadly defined, with examples including email, network, internet, file-system, and access control. Alternatively, coverage areas can be defined by breaking broader coverage areas into detailed fine-grained coverage for specific file-types, communication protocols, or system calls covered/monitored. The choice of level of detail for defining coverage areas will be motivated by the goal of the individual exploring an Effective Security-in-Depth problem. For an average non-technical user it may be sufficient to consider coverage as broad as what may be be covered by an common anti-virus suite. For a more technical user coverage areas may be more suitably considered as specific areas of network, email, and file monitoring. On the other hand, a security system designer may consider coverage as detailed as which specific kernel points or streams of data are being monitored.

- S^{avail} is the set of m available security mechanisms such that $S^{avail} = \{s_1, s_2, \ldots, s_m\}$. Each security mechanism $s_i \in S^{avail}$ can be considered as satisfying a subset of coverage areas, such that $s_i \subseteq C$.

  The areas of coverage for a security mechanism have been determined by either the mechanism's developer or external examination. Examples of security mechanisms include firewalls, virus scanners, spy-ware tool-bars, link scanners, phishing/spoofing plug-ins, spam scanners, intrusion detection systems, file monitors, and access

control systems. This list could also be considered to include any secondary programs that operate to maintain a security mechanism's functionality. The majority of modern anti-virus security suites consist of a subset of coverage areas. The goal is to find mechanisms that fulfill some desired depth of coverage.

- The desired level of security coverage D is a function $D : C \to \mathbb{Z}_{\geq 0}$ that produces the desired non-negative integer depth $D(c_i)$ for every coverage area $c_i \in C$.

  This desired depth of coverage is defined differently for users with different usage requirements. As an example, for one area of coverage c_i, a depth of $D(c_i) = 0$ would indicate that no layers are desired.

These formalizations allow for the definition of the coverage function which is utilized to determine if some prospective set of security mechanisms fulfills some level of desired coverage.

Definition 2.2.1. Coverage Function (cov)

The coverage function

$$cov : C \times \mathbb{P}\left(S^{avail}\right) \to \mathbb{Z}_{\geq 0}$$

determines the depth of coverage provided by the set $S^{?}$ of candidate security mechanisms for each area of coverage $c_i \in C$.[1] □

This coverage function allows for the definition of the Security-in-Depth problem.

Definition 2.2.2. Security-in-Depth (SiD) Problem

The SiD problem is the determination if a subset $S^{?} \subseteq S^{avail}$ of available security mechanisms achieves a specified desired depth D of coverage as evaluated using the coverage function cov. □

To determine if some set of prospective security mechanisms is a solution to the SiD problem, the coverage function is applied to each coverage area $c_i \in C$. If

[1] $S^{?} \subseteq S^{avail}$ is a set of security mechanisms. It has not yet been determined if this prospective set of security mechanisms is a covering set S^{cov} of security mechanism.

$cov\left(c_i, S^?\right) \geq D(c_i)$, then this indicates that $S^?$ provides the desired level of security coverage. This prospective set of security mechanisms $S^?$ can then be considered a covering set of security mechanisms.

Since there are likely many possible covering subsets S^{cov} that fulfill the requirements of desired coverage D, the set $Cover^D$ will be defined as the set of all covering subsets S^{cov} found in S^{avail} that provide the desired depth D of coverage (i.e. $Cover^D \in \mathbb{P}\left(S^{avail}\right)$). A set of security mechanisms $S^?$ that fulfills the desired coverage can then be considered $S^? = S^{cov} \in Cover^D$.

2.3 Effective Security-in-Depth

The Effective Security-in-Depth (ESiD) problem extends the base SiD problem to that of selecting a subset of security mechanisms that, in addition to fulfilling the desired depth of coverage, provides the desired functionality and minimizes performance loss. This performance/functionality is evaluated according to metrics observed via interactions with a collection of programs. These metrics and interactions are representative of the performance and functionality loads created by a user.

To definition of the ESiD problem requires formalizing the involved parts.

- P is the set of l programs in the system that can be interacted with such that $P = \{p_1, p_2, \ldots, p_l\}$. Each program $p_j \in P$ can be interacted with via a set of possible interactions $p_j = \left\{ip_{j,1}, ip_{j,2}, \ldots\right\}$.

  The set I^P contains all the possible interactions for the complete set of programs P. Example interactions include installing/starting/stopping a program, inputting parameters, or triggering an operation within the program. Some examples include copying files between local or network locations, compressing or extracting files using 7-zip, and encoding or decoding mp3/mpg files through video tools [LBS09].

- The metrics for the programs P are measured during the execution of interaction event sequences. An event sequence $es \in ES^P$ consists of a sequence of interactions, such that $es = (ev_1, ev_2, \dots)$. An interaction event ev is the execution of some program interaction $ip_{j,k} \in I^P$ at some time $t \in Time$ such that $ev = \big(ip_{j,k}, t\big)$.

  ES^P is the set of all possible interaction event sequences $es_i \in ES^P$ given the available interactions I^P. $ES^{\mathit{eff}} \subseteq ES^P$ is the subset of these interaction event sequences which are representative of the user's possible usage of the system, under which it should remain effective in terms of performance and functionality.

- The performance and functionality of the system during an event sequence are assessed according to a set $M = \{m_1, m_2, \dots, m_o\}$ of o metrics. Some of these metrics are general system metrics and are program independent, such as measures of system memory usage, processor usage, etc. However, in some cases a metric is associated with a specific interaction with a program.

- A metric $m \in M$ consists of a pair, such that $m = ([a, b], \mathit{sgn})$ with $a, b \in \mathbb{R}$ and $\mathit{sgn} \in \{-1, +1, N/A\}$ where $[a, b]$ represents a range of performance or functionality that is acceptable. A strict functionality metric will have $\mathit{sgn} = N/A$, while a performance-functionality metric will have either $\mathit{sgn} = -1, +1$.

  A value, $v_m(es) \notin [a, b]$, observed during the execution of event sequence es is unacceptable for a metric and would indicate a set of security mechanisms as being non-functional. Note, an a value of $-\infty$ indicates that there is no lower bound, and a b value of $+\infty$ indicates the same for the upper bound. A $\mathit{sgn} = N/A$ indicates a strictly functional metric, otherwise $\mathit{sgn} = -1, +1$ indicates that a better value for the metric is a lower numerical value with -1, or a larger value with $+1$.

There are two general classes of metrics: performance-functionality metrics and strict functionality metrics. Functionality metrics only measure the success and failure

of system operations, while performance-functionality metrics also measure the range of performance or costs to the system while operating.

1. A simple strict functionality metric is a measure of success or failure where $m^{succ} = ([1,1], N/A)$. Success is measured as a value of 1. However, a measured value of 0 indicates failure, since it is outside the acceptable range $[1,1]$. Being unable to complete a desired sequence of interactions is an example of a failure value of 0.

2. The simplest example of a performance-functionality metric is a measure of time for a sequence of interactions to complete. An example of this is a time metric m^{time} defined as $m^{time} = ([0,b], -1)$. The range $[0,b]$ is the acceptable functional duration. Taking an unacceptable length of time to complete the interaction would consist of observing a value $v_{m^{time}}$ for the metric such that $v_{m^{time}} > b$. Based on $sgn = -1$, better performance is to minimize this metric towards 0.

There are a couple of challenges in mapping real-world interactions to this formal representation. In the real world, an interaction event sequence is possibly infinite in length. It is also not possible to evaluate every one of the interaction event sequences found in ES^{eff} given the size of the set, even when limited to finite sequences. This motivates the development of the exploratory method described in Chapter 5, utilizing evolutionary search to discover interaction event sequences in ES^{eff} that are the most challenging to system performance. This methodology is designed to allow for the determination of whether or not a prospective set of security mechanisms is a solution to the Effective Security-in-Depth problem.

Definition 2.3.1. Effective Security-in-Depth (ESiD) Problem

The ESiD problem is the determination of an effective subset S^{eff} of security mechanisms. These security mechanisms are a covering subset for the desired depth of coverage D such that $S^{\mathit{eff}} \in \mathit{Cover}^D$. Additionally, the subset S^{eff} fulfills the desired performance-functionality predicate *perf* that mandates functionality levels, while minimizing the extent of unwanted performance loss in regard to the effectiveness measure *eff*. □

Functionality and performance are examined under a set M of program metrics measured during the execution of user interaction event sequences in $\mathit{ES}^{\mathit{eff}}$, which are representative of the user's performance requirements. There are likely many possible subsets S^{eff} that are solutions to the ESiD problem. The set $\mathit{Eff}^M \subseteq \mathit{Cover}^D$ is the set of all subsets S^{eff} of security mechanisms found in Cover^D that provide the desired performance and functionality of M.

The predicate *perf* is used to decide if a selection of security programs achieves the desired absolute level of functionality in regard to the metrics M for some interaction event sequence $es \in \mathit{ES}^{\mathit{eff}}$. This absolute value is a level of functionality that, if violated, represents a system state that is unacceptable to the user. There is an assumption that this predicate should always be satisfied by the system state protected by no security mechanisms $S^0 = \{\}$. In other words, it is desirable that S^0 should be such that *perf* is true for all event sequences in $\mathit{ES}^{\mathit{eff}}$.

Definition 2.3.2. Performance Functionality Predicate (*perf*)

The performance-predicate *perf* is a predicate where a passing positive answer is determined by examining if all observed values v_m of metric $m \in M$ fall within their acceptable range $[a, b]$ such that $a \leq v_m \leq b$. □

If all metrics are within the indicated acceptable range for the event sequences in $\mathit{ES}^{\mathit{eff}}$, then the goal is to minimize unwanted performance losses. That is, given

a performance loss effectiveness measure *eff*, the goal is to avoid event sequences in ES^{eff} that increase *eff*.

In general, each addition of another security mechanism to the prospective set of security mechanisms $S^?$ will have a negative effect on most performance measures examined for the system. However, an event sequence that creates performance load pressure on the secured system may equally create load on a completely unsecured system S^0. To differentiate, the effectiveness measure *eff* is a relative measure of the differential between performance for the system with some subset $S^?$ of security mechanisms installed and the baseline performance of the system S^0 where none are installed.

Definition 2.3.3. Effectiveness Measure (*eff*)
The effectiveness measure *eff* is the weighted summation of performance metrics M under the set $S^?$ of security mechanisms in regard to an event sequence $es \in ES^{eff}$ such that

$$eff\left(S^0, S^?, M, es\right) = \sum_{m_i \in M} Max\left(0, w_i \cdot diff_{m_i}\left(v_{m_i}(es), v_{m_i}^{base}(es)\right)\right)$$

where $w_i \in \mathbb{R}_{\geq 0}$ and $diff_m\left(v_{m_i}(es), v_{m_i}^{base}(es)\right)$ is a function defined for the metric m that returns the difference between the measured $v_{m_i}(es)$ and baseline $v_{m_i}^{base}(es)$ values for the metric m_i measured during the execution of the event sequences es. □

The differential function *diff* returns an increasingly positive value representative of the performance loss between the measured value and baseline values. The direction that the differential function *diff* considers performance loss is indicated by the sgn_m of each performance metric m. Negative differentials returning less than 0 are ignored, since it is likely that different metrics may be positively/negatively affected by the same sequence. It is desirable to avoid the differentials in such instances canceling each other out upon summation. Purely functional metrics are not included in the

effectiveness measure. The issues of system performance being dynamic between different executions of the same interaction event sequence will be addressed in the formulation of the definition of *diff* developed in Chapter 5.

The performance-functionality predicate *perf* and effectiveness measure *eff* are used to guide the developed exploratory evolutionary search. The performance predicate *perf* is used in the evolutionary search to indicate specific event sequences where the functionality or performance of a security mechanism combination $S^?$ was unacceptable. At the same time, the effectiveness measure *eff* is in the fitness function to direct the evolutionary search. Taken together, if a prospective subset of security mechanisms passes the *perf* predicate and does not exhibit unwanted levels of performance loss for explored event sequences in ES^{Eff}, then it may be considered as a candidate solution for the ESiD problem.

With the background and definition of Effective Security-in-Depth established, the following chapter will explore related works.

Chapter 3

Related Work

There is a selection of research areas relevant to the previously described problem of Effective Security-in-Depth. The most fundamental of these requires the discussion, in Section 3.1, of the challenging problem of measuring security. Collaborative security is the ideal solution for Security-in-Depth, and Section 3.2 explores research in that direction and explains how the developments of this thesis relate to this optimistic solution. Unified Threat Management (UTM), introduced in Section 3.3, is the best known, although limited, top-down application of collaborative security in which a selection of security mechanisms is designed by a single vendor to work together to provide network security for a system. Finally, the solution developed in this thesis uses heuristic search techniques to explore the performance capabilities of secured system configurations. Existing developed techniques for exploratory testing are discussed in Section 3.4.

3.1 Measuring Security

As Lord Kelvin once stated "If you cannot measure it, you can not improve it." Measurement remains one of the fundamentally hard challenges for security research [Bel06]. This is not because there is a lack of values within a computing system to measure, but because there is no precise way to quantify security. Useful verifiable metrics in security are few and far between, and there are many avenues of research remaining to be explored [SBE11].

Some attempts have been partially successful in measuring security coverage, such as with Relative Attack Surface Quotient (RASQ). RASQ quantifies the relative

attack-ability by describing potential points of exploitation and determining a relative vulnerability level based on real-world exploitations known or discovered [HPW05]. However, the methodology used is limited in that it requires an initial assumption that the points of security vulnerabilities and exploits are known or predictable. Although some security vulnerabilities may be predictable, this predictability is generally based on past knowledge of previously seen exploitations. Security vulnerabilities and the resultant exploitations are not enumerable, that is they are uncountable [Ros08]. There is no known definitive way of predicting what part of an individual code base, running program, or larger interacting system of programs may be exploitable.

The primary challenge to security is the ingenuity of the opponent. Without this ingenuity, all that would be necessary is to take into account past exploitations and make the necessary changes to counter them. Human opponents are well known for being smart, agile, cooperative, and able to create new tactics even as past exploitations are discovered and countered [Ros08]. The biggest advantage of the attacker is that defence requires the protection of every asset, while an attacker must only find a single vulnerability in the coverage. This asymmetric effort is Sun Tzu's advantage for the attacker, since it is the weakness of the defensive position in computing.

> "If they are substantial, prepare for them; if they are strong, avoid them."
>
> Sun Tzu - Art of War [SS94]

The smart attacker knows that there are almost infinite vulnerabilities and that they only need to succeed once if there is a single layer of coverage [Ros08]. The goal of the Security-in-Depth paradigm is that, if the attacker succeeds in bypassing a layer of coverage, then they encounter a subsequent layer to challenge them. In the ideal construction, this following layer does not come into play until the previous

is bypassed, optimistically introducing uncertainty into the attacker's plan [Sty04]. Many real-world Defence-in-Depth scenarios rely on making subsequent barriers a unique challenge from the prior barriers. For example, in trench warfare, it was common to layer wire entanglements, listening posts, a machine gun fire trench, a supervisor trench, trench mortars, and a rear support trench [Wal17].

One of the primary reasons for this organized layering is its robustness and that the addition of each layer is not simply additive but multiplicative. Multiplicative defence implies that each subsequent addition does not simply add its own difficulty, but that its interaction with the previous layers multiplies the challenge it provides [Sty04]. For example, attempting to overcome barbed wire entanglements after being spotted and put under machine gun and mortar fire is much more difficult in combination than each in isolation. Unfortunately, this is not a common feature of computing system security. Brittleness is the characteristic in computing system security, where security coverage layers are simply additive [Bel06]. One solution to brittleness is the idea of collaborative security where security mechanisms are designed to be cooperative and create the multiplicative effect. This is discussed in Section 3.2.

One noticeable challenge is that it is increasingly common that a vulnerability is not directly the fault of a single program, but is in fact emergent from the interactions of a larger collection of programs [Bel06, Mog06]. Anti-virus systems are often incompatible due to the conflicts caused by their indiscriminate modification of system properties and kernel data structures [Sky06]. Despite this challenge, the goal of Security-in-Depth is to use such a larger collection of security programs to provide layered security. This is a fundamental difficulty for all complex security systems, since it is not possible to understand the limitations of layered security, or how the interactions between the programs forming each layer may introduce new vulnerabilities [TFB10].

It is important to state that it is a myth that more layers of defence are always better than fewer [TFB10]. Just because one security mechanism may be good, and two security mechanism may be better, it is not true that using *N+1* security mechanisms is better than using N. Even on the modern desktop, where a user may have unused storage memory, operating memory, and processor time, each security mechanism alters how the system performs. This is particularly noticeable due to the necessary design problem that in order for a security mechanism to monitor the system it must attach itself to system calls, monitoring points, and performance operations. In many cases the conflict of these attachments and monitoring is more than additive by itself when the security mechanisms compete for resources [LBS09].

This thesis does not tackle the challenge of determining the coverage of a security mechanism. The determination of coverage is by itself a difficult problem still under active research. The claimed security coverage of a mechanism is accepted at its evaluated or claimed level. The goal of this thesis is framed from the end user's perspective, where there are a number of possible security mechanisms available and the goal is to select the most effective combination of them within their claimed coverage abilities. Similarly, the possibility of there being multiplicative value to certain combinations of security programs is not quantified. This is partially because many available security programs are not designed for collaborative operation, and because such collaboration is currently under study and largely difficult to quantify. Such collaborative security research is discussed in the following section.

3.2 Collaborative Security

Collaborative security, as briefly discussed in the previous section, is constructed based on the concept that Security-in-Depth is most effective when security mechanisms inter-operate to provide multiplicative, rather than additive, layers of security.

In some cases, it is desired that this collaboration begins with simply allowing installation at the same time, let alone simultaneous operation. For example, many anti-virus systems either warn of the need, or require, the uninstallation of other vendor's anti-virus suites [LBS09]. The most common experienced form of security collaboration is generally found in network security, where it is no longer surprising to see vendors offering network security devices that combine firewalls, intrusion detection/prevention systems, honey-pots, and vulnerability scanners [MBA09, CBJW09]. These top-down collaborative security systems are often called Unified Threat Management (UTM) systems, and they are discussed in the following section.

Collaborative security ideas are not necessarily limited to the technology side of protection. Intel's Defence-in-Depth strategy utilizes both personnel and computing resources collaboratively [Ros08]. Security-in-Depth coverage is provided by technology, but it is monitored by prediction teams who attempt to detect attackers, repair the system, predict future similar attacks, and design preventative measures for that goal. The importance of this complete Defence-in-Depth strategy is to design the technology side to both protect and provide valuable information to the prediction teams. This process is motivated by the understanding that deterring or preventing attacks is a better return on investment than the cost of responding to successful attacks [Ros08].

In the area of collaborative security research, most solutions are often focused on collaboration within a single field of security coverage. For example, within the field of firewall security identification and resolution, firewall policy rule conflicts in single and multi-firewall environments could be considered a collaborative security design [ASHBH05]. Another example is conflict resolution between firewalls, intrusion detection systems, and network access control lists [YMS+06]. These three areas of security are strongly related in the field of network security, where network connec-

tivity is affected by firewall interference in network address translation, access list network authentication, and anomaly detection by intrusion detection systems. As a result, many collaborative security solutions, such as those found in UTM research, are limited to a single coverage area or related subset of coverage areas.

Collaborative security methods have been developed to deal with internet worms. Autograph accomplishes this by analysing the prevalence of portions in flow payloads [KK04]. The first thing Autograph does is to automatically generate worm signatures for new internet worms propagated via TCP transport. Then Autograph uses an automatic worm defence method such as Vigilante to perform collaborative worm detection on end hosts. In Vigilante, hosts broadcast self-certifying alerts on worm detection, which are used by other hosts to generate filters to block infection [CCC+05].

Unlike the previous methods, which define collaboration as being across a network of computers, Sweeper is a malware solution that is defined within a single system. Sweeper uses lightweight monitoring techniques to detect suspicious requests, which are then supplemented by heavyweight monitoring when attacks are detected [TNL+07]. From the heavyweight analysis, anti-bodies are produced within the system to prevent future attacks against the same vulnerability. These anti-bodies are either input signatures or more robust Vulnerability Specific Execution Filters that identify an attack for future prevention. Sweeper also enables the roll-back of system state to recover from infections.

Another security method that attempts to repair buggy programs through collaboration can be found in software self-healing [LSK06]. The method is built around the concept of Application Communities as a collection of independent instances of the same application spread across the network. Each application is responsible for monitoring for attacks and notifying the remainder of the community. The mitigation

method of Selective Transactional EMulation (STEM) is used to both monitor and provide re-mediation for discovered low-level software faults [SLBK05]. This method again relies on a network of computers to host application communities of sufficient scale.

Within the research area of collaborative security, solutions consisting of a single computing system are uncommon. This is especially true when the collaborative security is desired across the complete spectrum of coverage areas. The most challenging aspect of this single computing system with broad spectrum coverage problem can be termed bickering. Bickering, as defined in [LBS09], is the challenge caused by the fact that security mechanisms are infrequently designed to co-exist, let alone interoperate. The mechanisms compete over hooking system calls, monitoring points, and other policy events. The motivating paper that defines the bickering problem also proposes that security mechanisms need to be developed in such a way that they collaborate in order to ease the management and performance burdens of running multiple security mechanisms.

One of the key issues in broad spectrum collaboration is trust management [BFL96, LGB10]. Trust management is again a difficult challenge in the security field. Especially given the recent confirmations that even the more respected state actors, such as the United States of America's National Security Agency, have been accused of influencing the introduction of vulnerabilities during the development of security mechanisms [Coh13]. Any broadly accepted trust management system would likely be a victim of a focused effort by such state actors to compromise the system.

This thesis uses a formulation of the Security-in-Depth problem that allows security mechanisms to be designed for collaboration. However, given the challenges facing trust management and collaboration, security mechanisms are evaluated based on their individual claims of coverage. Additionally, the problem formulation is struc-

tured to deal with the challenge that security mechanisms may not fulfill their claimed coverage or may even be compromised in some manner. The purpose of the multiple layers of coverage, within a particular coverage area of Security-in-Depth, is to counteract this possibility.

3.3 Unified Threat Management (UTM)

Intrusion Detection Systems (IDS) are often considered state of the art in network protection [Den87]. IDS and their active variant Intrusion Prevention Systems (IPS) are used to identify anomalous and dangerous network behaviour and, in IPS, implement measures to prevent it. The ability to recognize intrusions within network behaviour is accompanied by a pattern creation and matching problem suited to artificial intelligence solutions [MPB+16]. Traffic type identification and classification of encrypted packet data has been assisted by using multi-objective genetic algorithms to determine feature selection and cluster count for K-means classification [BZHH11].

Recently, IDS research has been incorporated in the top-down collaborative security designs known as Unified Threat Management (UTM) systems [HMF12]. UTM systems are generally created by a single vendor combining a number of hardware, software, and network-layer technologies such as firewalls, pattern recognition engines, and IDS [CBJW09]. This single-vendor approach allows for the top-down design methodology and the enforcement of the desired collaboration. UTM systems operate as a combined security gateway between the greater Internet and the local enterprise environment [DLY+08].

The layering of UTM systems is designed to make the security problem more manageable. For example, the firewall acts as a primary filter and barrier to the most obvious malicious activity. However, the integration of what were previously more narrowly focused technologies makes the management of UTM systems challenging

[ZDC+10]. Creating a UTM requires the integration of different control interfaces, message formats, communication protocols, and security policies. IDS are still active within the greater UTM environment using different techniques, including protocol analysis, signature-based detection, anomaly detection, and intrusion-detection sensors to identify network threats [MHL94].

Security-in-Depth forms a critical design element of the UTM architecture. This Security-in-Depth is accomplished in a variety of manners. Sometimes it is within the UTM system itself, but more often it is by layering multiple UTMs, each often using different technology than the previous one. Another Defence-in-Depth technique is the honey pot technique. A honey pot is a faux program, system, or environment into which suspicious activity is directed for monitoring [TFY03]. The honey pot technique is often employed by security researchers and experts to monitor and explore newly discovered or unsolved malicious activities. Some important benefits of the UTM combination of hardware, software, and networking technologies include cost reduction with fewer network monitoring devices and reduced device management, easier configuration, and in many cases network attacks are stopped before they enter the local network to attack individual machines [CBJW09].

Unified Threat Management systems, despite being limited to the singular coverage area of network security, are often the victim of the performance problem addressed in this thesis. In [DLY+08], Deng et al. explore how UTM systems do not guarantee network performance, and in many cases actually slow network activity. This problem is the result of the inter-operation of the many different layers, which increases the time to gather and process network data. The more independent the layers of the UTM are, the slower the network throughput is, since data packets are delayed during processing, thus bottle-necking the network [QYXL07, HY08]. It is possible that the configuration of a UTM system could benefit from the devel-

opments of this thesis. The choices of which components and operational balance of monitoring and prevention could be explored by the developed testing methods. This is because the UTM system design problem could be considered a subset of the Security-in-Depth problem, in which the desired coverage is limited to network security.

3.4 Adaptive Exploratory Performance Testing

The search space of possible representative operational loads that a user may place on the secured system is both expansive and diverse. There are innumerable combinations of programs, and interactions with such programs, which may take place in the infinitely long sequences of interactions that represent user load. This is a challenging, but not necessarily unique, set of problems that must be addressed by a search algorithm chosen to explore this combinatorial search space. The field of intelligent search is populated by the task of properly fitting search algorithms to challenges and characteristics of the application areas.

It is not feasible to explore the search space described above using an exact search algorithm. An exact search algorithm is guaranteed to find the optimal solution, as defined by a particular search goal, in a given search space. However, the size of this non-deterministic polynomial-time hard problem means that an exact search algorithm may take an unacceptably long period of time to finish. The alternative, which this thesis will use, is to use a heuristic algorithm that, in exchange for taking an acceptable period of time, finds only a rather good solution while relinquishing the promise of finding the optimal. Some possible heuristic optimization algorithm types include genetic algorithms [Gol89, Mit96], simulated annealing [KVG83], tabu search [GL03], particle-swarm optimization [KE95], and ant methods [DMC96].

This thesis is concerned with exploring the possible space of representative oper-

ational loads given a set of security mechanisms installed on a system. Such hard to define explorative testing is often considered the responsibility of human-driven manual testing. In manual exploratory testing experienced software testers are tasked with developing these different possible operational loads [KFN93]. The most basic form of this testing could be considered the realm of benchmarking, where a person or group has determined an operational load designed to demonstrate the differentials between different software for a given metric [Fut13, BAP13]. However, benchmarks are just static tests and not truly exploratory. The software tester in exploratory testing is further tasked with learning during the process of testing to progressively design higher quality tests that expose new deficiencies in the tested system.

The set of all possible tests applicable to a system can be considered the search space for an optimization problem. The job of the exploratory tester is to explore within this search space. The optimization goal of the exploratory tester is to create a suite of tests that find deficiencies in the tested system. Exploratory testing is both random, when the tester explores different areas of the system's operation, and heuristic, when the tester acts on intuition and past knowledge to explore areas considered the likely source of deficiencies. It is clear that exploratory testing shares the characteristics of heuristic search. This thesis is in effect the development of a novel exploratory testing method and tool for the unique problem of Effective Security-in-Depth.

This thesis leans on the demonstrated success of applying genetic algorithms to an exploratory testing problem. A genetic algorithm is a type of heuristic evolutionary optimization algorithm based on the principles of the biological evolution of a population of individuals. The possibilities of evolution, given the encoding (DNA) of an individual, define the search space that a genetic algorithm searches within for the most fit individuals. Biological principles of the survival of the fittest, natural

selection, and the mutation and recombination of genetic information, are used to explore this search space [Hol75, DJ75, HH98, Gol89]. Crossover distinguishes genetic algorithms from hill-climbers and simulated annealing by allowing the sharing of evolutionary information between different individuals of the population. Genetic algorithms generally deal well with the "*exploration vs. exploitation*" problem of balancing the finding of new local optima with focusing on found local optima for the global optimum [Hol98]. Genetic algorithms are also applicable to multi-objective search problems, where consideration must be made for the minimizing/maximizing of multiple interacting variables [HH98].

This research makes use of a fitness function that combines multiple weighted metric measures together into one fitness function. The created genetic algorithm therefore does not fall within the definition of a standard multi-objective genetic algorithm that is capable of exploring the Pareto optimal front for multiple fitness objectives simultaneously [HH98]. When using multiple metrics the methodology developed is not specifically interested in the maximized values of each metric. The method is directly interested in the examples of performance decrease or functionality failure presumed to be associated with these pressuring these metrics. Finding the Pareto optimal front itself has little value to the current specific search goal. However, other applications of the general methodology introduced may find the Pareto optimal front of more interest. An additional benefit, of using a traditional genetic algorithm over a multi-objective genetic algorithm, is that the exploration process of the search explores and advances quicker through the search space as the result of a one objective limitation.

There are other areas of exploratory testing that are unique from the application area of this thesis but should be noted. A usage of Belief-Desire-Intention (BDI) agents to perform adaptive performance testing for web services is described in

[MCBH10]. The focus of this work is the design of test cases through the collaboration and communication of the BDI agents. The work of [GFX12] is concerned with performance testing via a feedback-directed black box software testing system. The Feedback-ORiEnted Per-fOrmance Software Testing (FOREPOST) method learns rules to select test data for future tests. This method relies on the formulation that all interaction with programs is via data and the method is not generalized for multiple applications. A process of using evolutionary algorithms for testing, but only for single autonomous agents (a self-organizing system), and limited to evolving positions of items in an environment space, is described in [NPT+09]. Additionally, a method for exploratory testing harbour patrol behaviour via particle swarm optimization is described in [FTD09].

One application of exploratory testing for a complex software system is found in [HDKB10, HDKB11, Hud11, HD15], where genetic algorithms search for sequences of events that produce undesirable inefficiencies in a dynamic and emergent multi-agent system. These sequences of events are structurally similar to the interaction sequences found in the Effective Security-in-Depth problem. However, in this thesis the relationship between interaction events and agents (programs) is more complex and the search goal is similarly unique. The variance, or noise, observed in the simulation and measurement process is also a greater factor. Another testing method with a complex system with emergent behaviours can be found in [BD13] where a learning evolutionary method is used to test precision agriculture sensor networks for exploitable vulnerabilities in the employed network protocols.

The application of exploratory search for performance testing for the problem of Effective Security-in-Depth is a new and unexplored area of research. Genetic algorithms have demonstrated utility and flexibility for providing an exploratory testing solution to problems with similar search space characteristics. This thesis provides a

genetic algorithm exploratory search methodology, a tool implementing this methodology, and the associated exploratory search processes to make use of the tool for the problem of Effective Security-in-Depth. Before each of these three contributions are explored, the following chapter will establish the definitions of the utilized evolutionary search method known as genetic algorithms.

Chapter 4

Introduction to Genetic Algorithms

Genetic algorithms are evolutionary algorithms inspired by the principles found in the biological evolution of a population of individuals. Each solution within the search space is transformed by encoding it as the DNA of an individual. A population of individuals is maintained and evolved. The genetic algorithm attempts to propagate the traits of the fittest individuals with the end goal of finding the most fit individuals (solutions) in the search space. Biological principles including the survival of the fittest, natural selection, and the mutation and recombination of genetic information, are used to explore this search space [Hol75, DJ75, HH98, Gol89].

Genetic algorithms are a search technique that cannot guarantee that the optimal solution in the search space will be found. However, they generally find a rather good solution in a reduced period of time. This is unlike exact search algorithms, which carry the guarantee of optimality, but for large NP-Hard problems generally take an unacceptable period of time. Like genetic algorithms, there are other search algorithm types that make similar compromises including simulated annealing [KVG83], tabu search [GL03], particle-swarm optimization [KE95], and ant methods [DMC96].

The Effective Security-in-Depth problem for event sequences of indefinite length is intractable, similar to many infinite search space exploratory search problems. When formally defined with a constrained upper limit of event sequence length and specific set of program interactions, this search space is suited for good enough search algorithms such as genetic algorithms. Chapter 5 introduces an application of genetic algorithms to the Effective Security-in-Depth problem, which makes use of the definitions described in this chapter.

Genetic algorithms have advantages that include:

- Exploring the given search space landscape in multiple directions at once via a population of individuals that can diverge.
- Avoiding local optima via population diversity and other methods. Local optima can easily trap other search algorithms such as hill-climbers.
- Sharing of evolutionary information between different individuals of the population via crossover operation. This is something hill-climbers and simulated annealing are unable to do.
- Balancing the "*exploration vs. exploitation*" problem between found local optima and the search for the global optimum [Hol98].
- Minimizing/maximizing of multiple interacting variables as found in multi-objective search problems [HH98].

There are challenges to designing a genetic algorithm as well:

1. The biggest challenge is determining the representation of solutions within the search problem as individuals. All valid solutions to the problem must be possible within the definition of an individual's DNA. The search landscape representation should be robust, such that individual solutions are reachable from similar individuals without separations of undefined space. Every individual in the search space need not be a valid solution. Sometimes it is necessary to make a trade-off of allowing invalid solutions that enable transitions to otherwise isolated solutions within the search space.
2. The next challenge is the creation of the fitness function. The fitness function represents a definition of the biological idea of what is considered "*the fittest*"

individual in the search space. Some problems have an inherent fitness function, such as a simple gradient performance measure of the level of success. The biggest difficulty is quantifying the separation of individual solutions' qualities. Gradations of improvement as an individual gets closer to optima are important because the fitness function guides the search.

3. There are a number of parameters that control a genetic algorithm. The most common of these are the size of the population and the rate at which different methods are chosen to create new individuals. There are also structural choices, such as if a population is replaced generationally or replaced piecemeal, or what type of selection process is chosen to pick fit individuals out of the population to be parents of the next generation.

4. Some problems will always be better suited to other algorithms. There are problems simple enough, or small enough, that an exact algorithm may be more than suitable. In such cases, it is impractical to implement a genetic algorithm when a result can be achieved in a more efficient manner with the guarantee of optimality.

5. If is generally infeasible to determine termination for many interesting genetic algorithm applications, due to in the challenge of not being able to determine if an arbitrary current solution solution is the optimal. In many cases, such as that in this thesis, the solution is to choose a length of time, or preferably iterations for generations of search populations, at which to terminate.

4.1 Individual and Fitness Function

The first step for using a genetic algorithm is to determine the representation (encoding) of a solution as an individual.

Definition 4.1.1. Individual (*indi*)

An individual *indi* is an encoding of a point within the search space into a representation as a sequence of DNA.

$$indi = (dna_1, \ldots, dna_i, \ldots, dna_p)$$

This encoding can be manipulated to reach other points within the search space. □

Each individual is a member in the set *Indi* of all possible individuals. *Indi* represents the search space of the problem as defined in the context of the search algorithm. The individuals that represent the current iteration (generation) within the search is a set *pop* of individuals known as a population. The set of all possible populations *Pop* is a subset of the power set of possible individuals, such that $Pop \subseteq \mathbb{P}(Indi)$. A search begins with an initial population, which is then followed by a sequence of subsequent populations until the search process completes. Figure 4.1 shows the relationship between an individual, population, and the generations of a search..

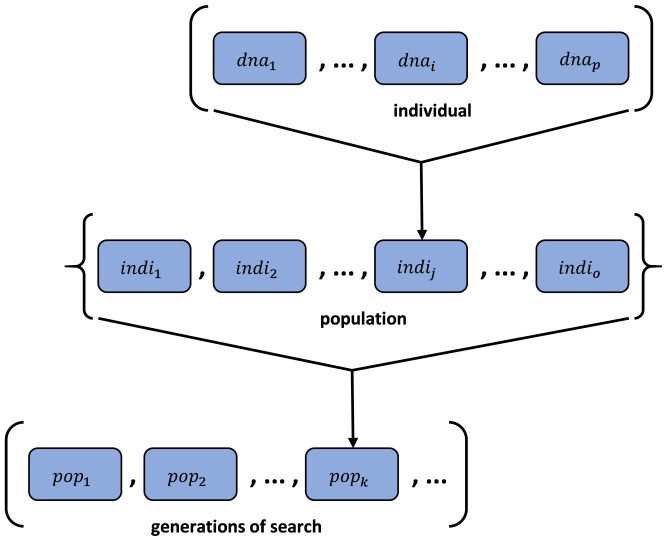


Figure 4.1: Genetic Algorithm Individual to Generation Relationship

A genetic algorithm attempts to evolve the individuals within its population over each generation to produce individuals that are as fit as possible. A fitness function

is required to quantify and measure this distinction. The search algorithm will treat either minimal or maximal values produced by this function as the most fit depending on the application area.

Definition 4.1.2. Fitness Function (*fit*)

A fitness function *fit* is a function that quantifies the fitness of an individual *indi*

$$fit : Indi \rightarrow \mathbb{R}$$

□

4.2 Search Model and Genetic Operators

With the individual representation and fitness function defined it is possible to define the generic search definitions used by genetic algorithms. First is the definition of the genetic algorithm (GA) search model.

Definition 4.2.1. Search Model (*Mod*)

The search model *Mod* is defined following the generic definition $Mod = (Pop, Tran)$ where

- *Pop* is the set of possible populations of the search.
- $Tran \subseteq Pop \times Pop$ is the transitions between populations with $(pop_i, pop_j) \in Tran$ being a transition from pop_i to pop_j in *Pop*. □

Genetic operators are used to define these transitions between population states. In general, a genetic operator is a function that takes the current population and generates an individual that becomes part of the subsequent population. Multiple different operators are utilized each time a new population is created.

Definition 4.2.2. Genetic Operator(GA_{OP})

A genetic operator GA_{OP} is a function

$$GA_{OP} : Pop \rightarrow Indi$$
$$GA_{OP}(pop) = indi$$

where a selection process $sel_{proc}(pop)$ is used to choose some number of individuals from pop that are altered and combined by a genetic operator to form a new individual $indi$. □

Generally, sel_{proc} is biased towards selecting the most fit from the population. However, most sel_{proc} methods are a variation on the concept of fitness proportionate selection where selection is stochastic but biased towards returning fit individuals. Example selection methods include: elitist selection [Thi98], roulette-wheel selection [Gol89], and rank selection[Gol89].

There are a number of standard genetic operators that can be defined for a genetic algorithm with a search space as defined. More specifically, the best operators op_{best}, survival operator op_{surv}, single-point mutation operator op_{mut}, and two-point crossover operator op_{cross}.

Definition 4.2.3. Best Operator (op_{best})

A best operator op_{best} is a genetic operator that ensures that the most fit individual remains in subsequent populations

$$op_{best}(pop) = indi_{best}$$

where the individual $indi_{best} \in pop$ has the best fitness found so far by the genetic algorithm. Note, this is a form of elitist selection. □

Definition 4.2.4. Survival Operator (op_{surv})

A survival operator op_{surv} is a genetic operator that allows for fit individuals to possibly survive into subsequent populations

$$op_{surv}(pop) = indi_{surv}$$

where $indi_{surv} \in pop$ is a fit individual from the previous population selected using sel_{proc} to survive into the next population. □

Definition 4.2.5. Single-Point Mutation Operator (op_{mut})

A single-point mutation operator op_{mut} is a genetic operator

$$op_{mut}(pop) = op_{mut}(indi_{parent}) = indi_{mutation}$$

where a single parent individual $indi_{parent} \in pop$ is selected by sel_{proc}, and a random number of DNA locations dna_i are mutated by being replaced with new dna_i^{mut}. This results in the mutated individual $indi_{mutation}$. □

The single-point mutation operator is depicted in Figure 4.2.

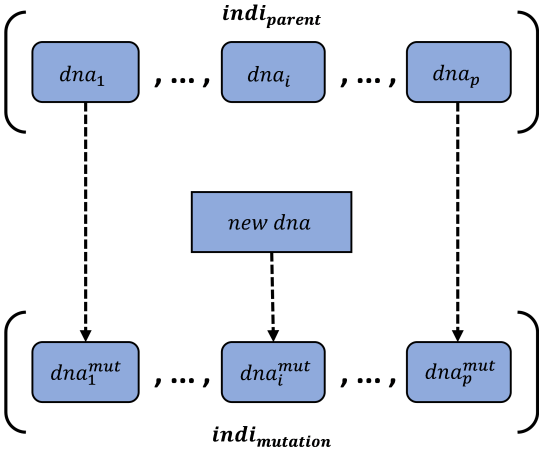


Figure 4.2: Genetic Algorithm Single-Point Mutation Operator

Definition 4.2.6. Two-Point Crossover Operator (op_{cross})

A two-point crossover operator op_{cross} is a genetic operator

$$op_{cross}(pop) = op_{cross}(indi^1_{parent1}, indi^2_{parent}) = indi_{crossover}$$

where two parent individuals $indi^1_{parent} \in pop$ and $indi^2_{parent} \in pop$ are selected using sel_{proc}. A sequence between the two indices m and n is filled with dna'_i from $indi^2_{parent}$ while the remainder is filled with dna_i from $indi^1_{parent}$. The result is the child individual $indi_{crossover}$. □

The two-point crossover operator is depicted in Figure 4.3.

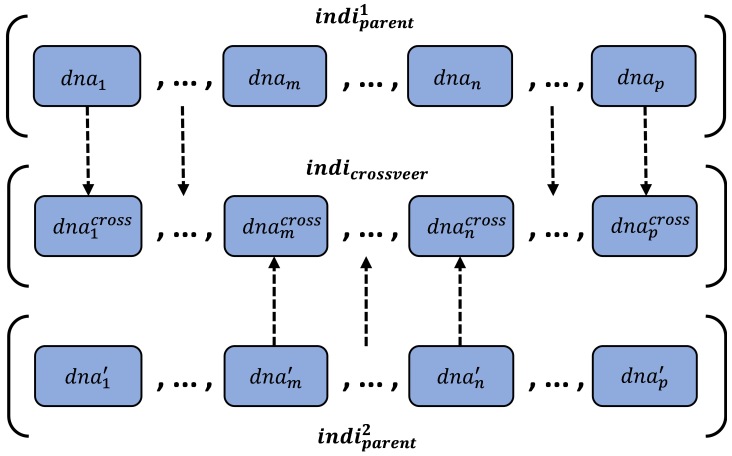


Figure 4.3: Genetic Algorithm Two-Point Crossover Operator

4.3 Search Control and Goal Condition

To this point the search space, individuals, population, fitness function, and transitions between populations via genetic operators have been defined. What remains is a definition of how the search process is controlled and determined to be complete. External input into the search processes is formalized by changes in the execution environment *Env*. *Env* allows for values to be communicated to the genetic algorithm, such as termination signals or external search guidance from other processes.

A genetic algorithm is directed by a search control, which is given configuration parameters to indicate what percentage of individuals for each generation are created using each genetic operator. There are also parameters that control the size of the population. A goal condition is used to determine when a search is complete.

Definition 4.3.1. Goal Condition (*Goal*)

A goal condition *Goal* determines if a current search state (population) represents the end of search.

$$Goal : Pop \to \{yes, no\}$$

Given that the search is generally unable to know if it has found the optimum (maximal/minimal fit individual) this limit is generally based on time, iterations, or properties of the population. □

If this goal does not occur, then values passed through the environment *Env* of the search process are generally used to terminate the search. After termination of the search, the result of the search is often considered the most fit individual discovered in any generation's population.

Definition 4.3.2. Search Control (*Contr*)

A search control *Contr* is the chosen method of selecting genetic operators and selection process sel_{proc} according to the fitness function *fit* to transition the population between states.

$$Contr : Pop \times Env \to Pop$$

$$Contr(pop, e) = pop'$$

where

- $(pop, pop') \in Tran$ is a transition between populations.
- *Env* allows for environment values to be communicated to the genetic algorithm.

□

A genetic algorithm search is generally initialized with a set of randomly generated

individuals. However, this population can sometimes be restarted from a previously developed population. In some other situations, the initial population is seeded with either previously found solutions, or random solutions with previously found subsequences.

The search control transitions the genetic algorithm through a sequence of population states. If no population state is defined as fulfilling the goal function, then this sequence is infinite unless terminated via another method. Traditionally, the result of a genetic algorithm is considered the fittest solution, as indicated by the fitness function, found at any point during the search. However, for methods such as exploratory search, there is information that can be taken as an aggregate from an examination of what was produced while exploring the search space. For example, event sequences that exhibit functionality failure may not be the fittest individual, but are of obvious interest as a result from the exploratory search. Given the stochastic nature of the sel_{proc} and operators, there are many different sequences of populations possible from the same initial random population. Instead of relying on a single search sequence, it is also possible for information to be gained from comparing the results of multiple search executions.

This formal definition is the framework in which the following chapter will define the specifics of a genetic algorithm for the problem of Effective Security-in-Depth. This definition does not cover all genetic algorithm variations, such as distributed or hybrid genetic algorithms. However, it is the foundation for the variant of genetic algorithm used in this thesis.

Chapter 5

Evolutionary Search for Interaction Event Sequences

This chapter introduces the abstract methodology of an evolutionary search for interaction event sequences for the problem of Effective Security-in-Depth. Most importantly, this chapter describes the construction of a fitness function to direct this evolutionary search. Effective Security-in-Depth was defined in Chapter 2, while Chapter 4 introduced the basic definitions required for a genetic algorithm. This chapter is built on top of this previous material, expanding on these general definitions with an individual definition in Section 5.1, specific genetic operators in Section 5.2, as well as a fitness function for the problem of Effective Security-in-Depth in Section 5.3. A simplified example continues throughout these three sections as a demonstration of the abstract definitions.

5.1 Individual Definition

For the problem of Effective Security-in-Depth, the search space is the set of possible interaction event sequences that can be created for a set of programs P. This set was introduced in Chapter 2 as ES^P. An individual solution is an event sequence $es \in ES^P$ made up of events $es = (ev_1, ev_2, \dots)$. Each event $ev = (ip_{j,k}, t)$ defines the execution of an interaction event $ip_{j,k} \in I^P$ for interaction k of program p_j at some time $t \in Time$. I^P is the set of all interaction-events.

The goal of the exploratory evolutionary search is to find event sequences that demonstrate that there are significant negative consequences to performance or func-

tionality as the result of installing some candidate set $S^?$ of security mechanisms. Every interaction has performance consequences even in the baseline system S^0. Therefore, performance consequences are evaluated to assess if the impact on metrics is greater for the secured system $S^?$ versus the baseline system S^0. During the search process, the expectation is event sequences that increase the pressure on the system metrics, and ultimately produce functionality failure or unacceptable performance, will be explored.

However, this means that every event sequence to be examined will be executed for both systems and judged comparatively. There is limited control over when actions will occur in the evaluated system. The response delay between the baseline system S^0, and the system with security mechanisms $S^?$, means that indicating a specific time value t for each event is not comparative between the baseline and a secured system. Instead of using time values, the sequence of events will be considered as an ordered sequence of event triggers with additional space between events represented by an additional *nop* (no-operation) event in I^P. The *nop* event executes a delay of a small period of time before proceeding to the next event.

This does not mean that the consequences of triggering an event will not overlap. It just means that the initial triggers of an event will not be simultaneous. It can be expected that event sequences in which all interactions are triggered at the same time would have caused the most resource consumption, while being a less realistic representation of a user's experience. If such sequences were allowed, then the search algorithm would converge on producing such examples and less likely to explore the type of event sequences that are the goal of the search. Resultantly, strict timing is ignored. Instead the concern is the ordering of a sequence of events rather than the relatively uncontrollable spacing between them.

From these definitions, with the removal of the consideration of $t \in Time$, the

result is an event sequence es of size p:

$$es = (ev_1, \ldots, ev_p) = (ip_{j_1,k_1}, \ldots, ip_{j_p,k_p})$$

This can be simplified into a sequence of integers where each j_i references a program in the set P of programs and each k_i an interaction for that program.

$$(ip_{j_1,k_1}, \ldots, ip_{j_p,k_p}) = (j_1, k_1, \ldots, j_p, k_p)$$

Definition 5.1.1. ESiD Individual ($indi^{ESiD}$)
An individual $indi^{ESiD}$ for the ESiD problem is a sequence of integers

$$indi^{ESiD} = (j_1, k_1, \ldots, j_p, k_p)$$

where $j_i \in \{1, \ldots, |P|\}$ and $k_i \in \{1, \ldots, |p_{j_i}|\}$ for each $p_{j_i} \in P$. □

For the Effective Security-in-Depth problem, the set of all individuals $Indi^{ESiD}$ is in fact a transformation of the set of all event sequences ES^P using the encoding of event sequence solutions into individuals.

The relationship between an individual, population, and the generations of a search can be seen in Figure 5.1.

5.1.1 ESiD Example

This subsection begins the simplified abstract example for the definition of an individual in an evolutionary search for the Effective Security-in-Depth problem.

There is one coverage area $C = \{c_1\}$, which will be considered as anti-virus coverage. A more detailed definition may prefer to more accurately define the set of coverage areas as a subset of more specific coverage areas generally found under an anti-virus system such as browser usage, email scanning, network monitoring, and file access monitoring.

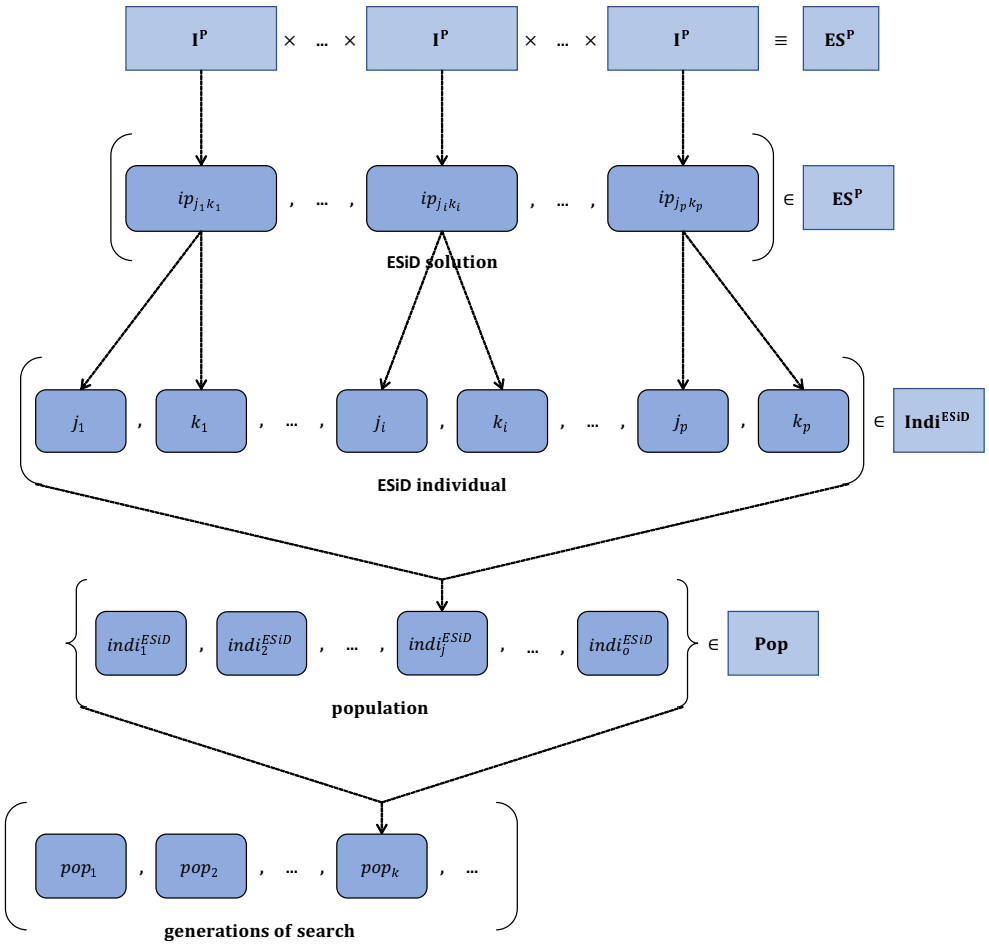


Figure 5.1: ESiD Individual to Generation Relationship

There is one security mechanism $S^{avail} = \{s_1\}$, such that $s_1 = \{c_1\}$ means the security mechanism provides anti-virus coverage. Therefore, the candidate set $S^?$ of security mechanisms consists of that single mechanism $S^{avail} = S^?$.

The desired level of security coverage is D, such that $D(c_1) = 1$, which means one level of anti-virus coverage is desired.

This means that the set $S^?$ of security mechanisms is a covering subset $S^? = S^{cov}$ of security mechanisms. This is true because the coverage function cov is such that $cov(c_1, S^?) \geq D(c_1)$. As a result, the prospective set $S^?$ of security mechanisms is a solution to the Security-in-Depth problem.

Now, to evaluate this set of mechanisms for the Effective Security-in-Depth prob-

lem, the parameters of the event sequence search space need to be defined.

There are two programs $P = \{p_1, p_2\}$. The first program p_1 is a zip compression utility, and p_2 is an encryption utility. Each program has two interactions such that $p_1 = \{ip_{1,1}, ip_{1,2}\} = \{zip, unzip\}$ and $p_2 = \{ip_{2,1}, ip_{2,2}\} = \{encrypt, decrypt\}$.

This means that the set I^P of all the possible interactions for the programs in P is $I^P = \{ip_{1,1}, ip_{1,2}, ip_{2,1}, ip_{2,2}\} = \{zip, unzip, encrypt, decrypt\}$.

An event sequence of length p is formed by selecting p interaction events from the set I^P. If $p = 3$, then two example event sequences are: $es_1 = (zip, crypt, zip)$ and $es_2 = (zip, decrypt, unzip)$. These event sequences can be transformed into their individual integer representation for the genetic algorithm.

$$es_1 = (zip, crypt, zip) = (ip_{1,1}, ip_{2,1}, ip_{1,1}) = (1, 1, 2, 1, 1, 1) = indi_1$$

$$es_2 = (zip, decrypt, unzip) = (ip_{1,1}, ip_{2,2}, ip_{1,2}) = (1, 1, 2, 2, 1, 2) = indi_2$$

One possible population $pop_i \in Pop$ of individuals for a generation i of the genetic algorithm may contain both of these individuals such that $indi_1, indi_2 \in pop_i$.

5.2 Genetic Operator Definitions

The initial evolutionary algorithm population consists of *pop_size* random individuals. Each random solution is formed from a sequence of stochastically chosen integer pairs. Each pair begins with the selection of a pseudo-random positive integer j from a uniform distribution in the size of P to indicate a program. A second integer k from a uniform distribution is then chosen in the size $|p_j|$ of that program's interactions to indicate which specific action will form that pair.

To create new individuals for subsequent generations to explore the search space, it is necessary to define the single-point mutation operator mut_{op}^{ESiD} and two-point crossover operator $cross_{op}^{ESiD}$ for the particular problem of Effective Security-in-Depth.

The ESiD single-point mutation and two-point crossover operators are depicted in Figures 5.2 and 5.3.

Definition 5.2.1. ESiD Single-Point Mutation Operator (op_{mut}^{ESiD})

The single-point mutation operator op_{mut}^{ESiD} is a genetic operator

$$op_{mut}^{ESiD}(indi_{parent}) = indi_{mutation}$$

where a single parent individual $indi_{parent} = (j_1, k_1, \ldots, j_i, k_i, \ldots, j_p, k_p)$ is selected by sel_{proc} and mutated at a random number of indices. The interaction event $ip_{j,k} \in I^P$ at each index i represented by j_i and k_i is replaced by $ip_{j^{mut},k^{mut}} \in I^P$ by substituting j_i^{mut} and k_i^{mut}. This results in the mutated individual $indi_{mutation}$. □

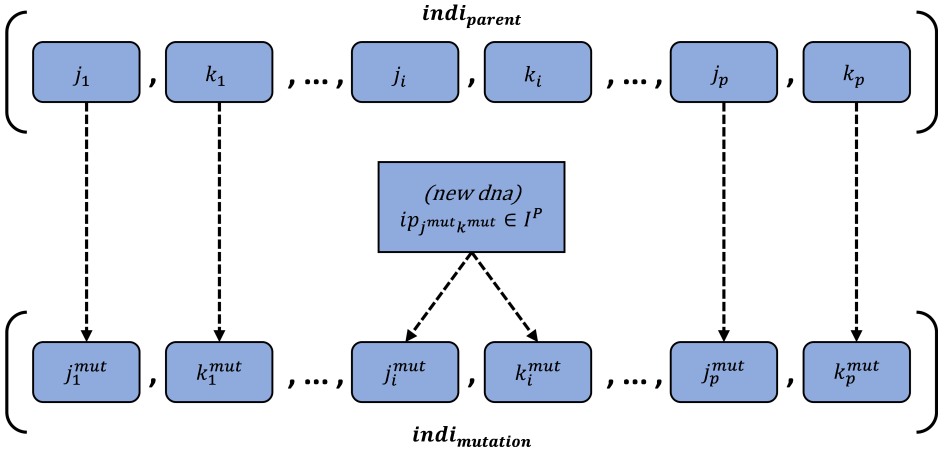


Figure 5.2: ESiD Single-Point Mutation Operator

Definition 5.2.2. ESiD Two-Point Crossover Operator (op_{cross}^{ESiD})

The two-point crossover operator op_{cross}^{ESiD} is a genetic operator

$$op_{cross}^{ESiD}(indi_{parent}^1, indi_{parent}^2) = indi_{crossover}$$

where two parent individuals $indi_{parent}^1 = (j_1, k_1, \ldots, j_p, k_p)$ and $indi_{parent}^2 = (j_1', k_1', \ldots, j_p', k_p')$ are selected using sel_{proc}. A sequence from the two indices m and n is filled with p_i' and k_i' from $indi_{parent}^2$ while the remainder is filled with p_i and k_i from $indi_{parent}^1$. The result is the child individual $indi_{crossover}$. □

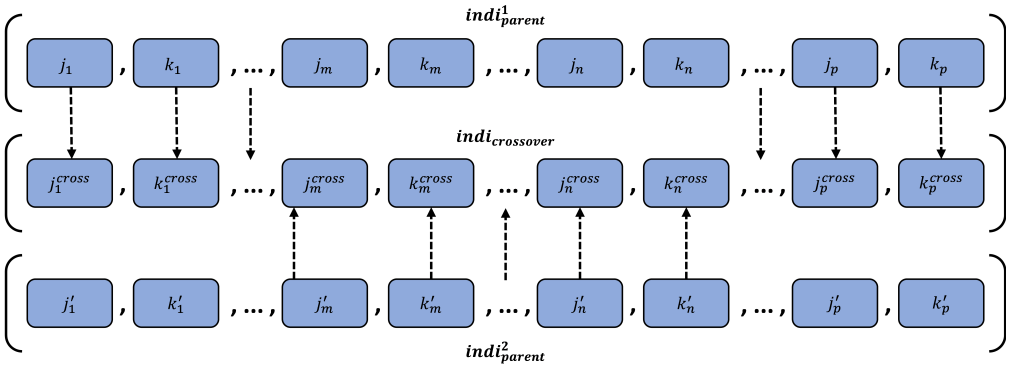


Figure 5.3: ESiD Two-Point Crossover Operator

The selection process sel_{proc} for choosing the parents for operators is a rank-based selection process based on the median fitness of a solution. Median fitness is used to avoid a singular extreme simulation result from biasing the selection. Using a median value results in solutions with more consistently high fitness evaluations being ranked higher. Rank-based selection orders the individuals in the parent population by fitness value, and biases selection based on how close an individual is to being ranked towards the top.

These are the fundamental operators used to create children for the next generation, however there are additional useful operators. One such additional operator op^{ESiD}_{best} maintains the previous best solution found. This operator is used once per generation, and simply determines the solution with the highest fitness in the previous generation, and promotes a copy of it into the following generation. There are two variants of this operator, one that propagates the individual with the highest maximum fitness seen, and one for the highest median fitness.

5.2.1 ESiD Example (continued)

This subsection continues the example of the previous section describing how operators create new individuals and genetic algorithm populations from previously existing ones.

The previous generation's population pop_{prev} will be considered to be that of pop_i given in the previous example. The current generation is created by the search process selecting genetic operators from the list given in this section. If the population size is o, then that count of genetic operators will be selected. Propagation operators such as op_{surv} and op_{best} will be used to move copies of individual from pop_i to pop_{i+1}. However, operators such as op_{mut}^{ESiD} and op_{cross}^{ESiD} will use the selection process sel_{proc} on pop_i to select parents to create child individuals for pop_{i+1}.

A single DNA mutation op_{mut}^{ESiD} example would select a single parent $sel_{proc}(pop_i) = indi_{parent} = indi_1$.

$$indi_1 = (1, 1, 2, 1, 1, 1) = (ip_{1,1}, ip_{2,1}, ip_{1,1}) = (zip, crypt, zip) = (ev_1, ev_2, ev_3)$$

To mutate this example, a single interaction event such as the first *zip* action ev_1, is selected stochastically. This *zip* action is replaced by a randomly chosen interaction event ev_{new} from I^p. For example, it will be replaced with *unzip*. The resultant child is $indi_{mutation}$.

$$(ev_{new}, ev_2, ev_3) = (unzip, crypt, zip) = (ip_{1,2}, ip_{2,1}, ip_{1,1})$$

$$(ip_{1,2}, ip_{2,1}, ip_{1,1}) = (1, 2, 2, 1, 1, 1) = indi_{mutation}$$

A crossover op_{cross}^{ESiD} example would select two distinct parents

$$sel_{proc}(pop_i) = indi_{parent}^1 = indi_1$$

and

$$sel_{proc}(pop_i) = indi_{parent}^2 = indi_2$$

.

$$indi_1 = (1, 1, 2, 1, 1, 1) = (ip_{1,1}, ip_{2,1}, ip_{1,1}) = (zip, crypt, zip) = (ev_1, ev_2, ev_3)$$

$$indi_2 = (1, 1, 2, 2, 1, 2) = (ip_{1,1}, ip_{2,2}, ip_{1,2}) = (zip, decrypt, unzip) = (ev_4, ev_5, ev_6)$$

A two-point crossover then selects two indices m and n. In this case, $m = n = 2$. The child individual is formed by selecting the start of the sequence from $indi_1$ up until index $m - 1 = 1$, then selecting the middle of the sequence from $indi_2$ between $m = 2$ and $n = 2$, and finally selecting the end of the sequence from $indi_1$ starting at $n + 1 = 3$ up to the end of the sequence. The resultant child is $indi_{crossover}$.

$$(ev_1, ev_5, ev_3) = (zip, decrypt, zip) = (ip_{1,1}, ip_{2,2}, ip_{1,1})$$

$$(ip_{1,1}, ip_{2,2}, ip_{1,1}) = (1, 1, 2, 2, 1, 1) = indi_{crossover}$$

These produced child individuals are accumulated to form pop_{i+1}. The search process then proceeds to evaluate these individuals using the fitness function before proceeding to form the next population pop_{i+2}.

5.3 Fitness Function

The final challenge for the search formulation is that of the fitness function. For the problem of Effective Security-in-Depth, the fitness function will be maximized. The desire is to find solutions that are examples of undesirable performance or functionality. However, not all solutions will necessarily finish successfully during an evaluation, and therefore result in no fitness. These individuals fail the Performance Functionality Predicate *perf*. If an individual passes the *perf* predicate, then its fitness can be evaluated using the Effectiveness Measure *eff*.

To form a fitness function, the desire is to make use of both *perf* and *eff*. To accomplish this, if *perf* fails (returns 0) for any one of the metrics, it should be possible to assign a base fitness value that distinguishes these individuals from those that pass (return 1). Those that pass will only be valued according to *eff*. If there is concern that valuing *perf* failures may be misleading, then it should also be possible to not quantify them in the fitness function.

Definition 5.3.1. ESiD Fitness Function (fit^{ESiD})

The fitness function fit^{ESiD} is a function that quantifies the fitness of an individual *indi* through a weighted combination of the Performance Functionality Predicate *perf* and Effectiveness Measure *eff*.

$$fit : Indi \rightarrow \mathbb{R}$$

$$fit(indi) = (1 - perf) * weight_{perf} + eff * weight_{eff}$$

□

Individuals that fail *perf* are of particular interest in many exploratory search goals. As a result, these are tracked externally to the main evolutionary search in a set *Fail*. These failures may not be repeatable, or common. In order to further examine these instances, an operator op_{failed} can be utilized to promote additional solutions that failed *perf*. There are two variants of this operator, one that propagates individuals that failed to complete the event sequence execution, and one that propagates individuals that created resulted in emulation disruptions during the management of the virtual machine that enables simulation of the event sequence.

An individual event sequence *es* is evaluated by executing itself once with the system configured as S^0, without any security mechanisms installed, and once as $S^?$, with the prospective set of security mechanisms installed. The fitness is then a measure of the difference in performance between these two configurations. A naive fitness function for an evolutionary algorithm would perform this execution once and consider the resultant effectiveness measure *eff* as a representation of the sequence's fitness. However, system performance is not static in nature and subsequent executions are unlikely to result in similar values. Instead of considering the first fitness value as final, each time a certain solution is executed its fitness is tracked. Extreme values are tracked by the search control to be examined by the user.

Similarly, the distribution of behaviour for metrics will vary between different

interaction event sequences and different prospective sets of security mechanisms $S^?$. To counter this, the definition of the differential function *diff* used in the effectiveness measure for each metric considers the distribution of the metric. Before the initial population of individuals is evaluated, a sample set of individuals is generated, and the values produced by it are considered a sample for each metric. In fact, the values produced with S^0 are considered one sample for the baseline system, and those for $S^?$ a second sample for the examined security mechanisms. Then, as subsequent generations are encountered, the differential function *diff* references the sample mean μ and sample standard deviation σ of the distributions for a given metric.

Figure 5.4 visualizes how the samples for the metric distributions used in the fitness function and differential function are generated.

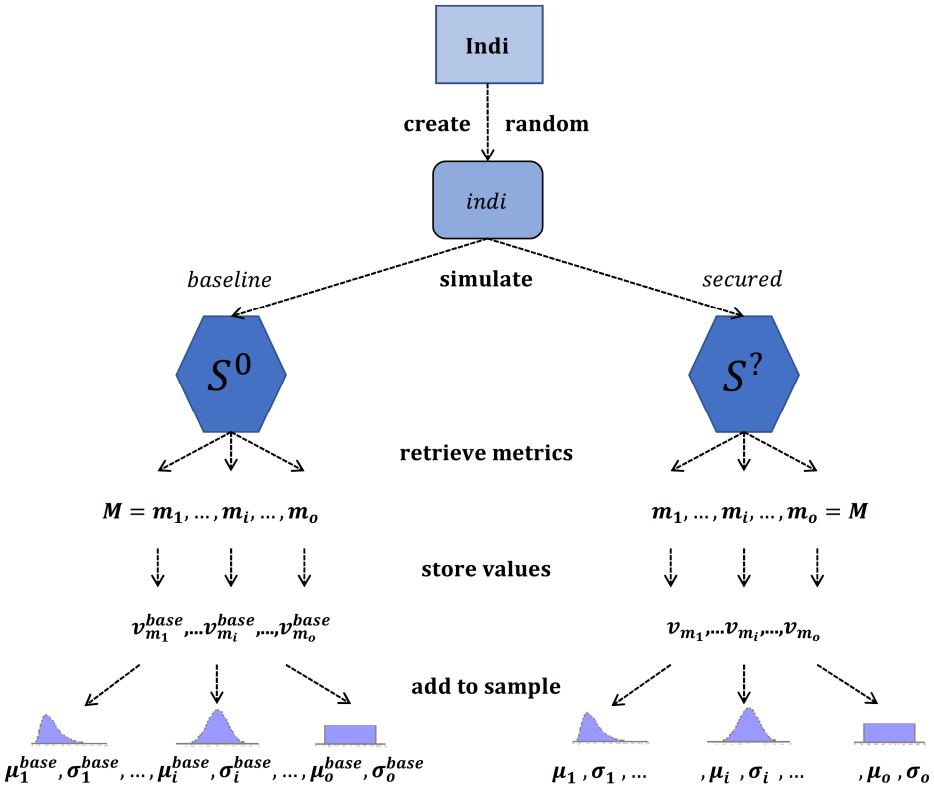


Figure 5.4: Metric Sample Distribution Creation

Definition 5.3.2. Differential Function (*diff*)

The differential function *diff*, for some metric m used in the summation of the effectiveness measure *eff*, is defined as

$$diff(v(es), v^{base}(es)) = -sgn \cdot \left[\left(\frac{v(es) - \mu}{\sigma}\right) - \left(\frac{v^{base}(es) - \mu^{base}}{\sigma^{base}}\right)\right]$$

where *base* values are the baseline measured for the unsecured system S^0 and the rest are those measured for the system $S^?$ for the event sequence es. The sign value *sgn* is a sign multiplier for the metric.

The result of this formulation is that values that represent worse performance than the baseline result in positive differential values. For metrics where $sgn = -1$ worse is a more positive value. Effectively, the differential value is a comparison like the statistical z-scores between the two configurations. A z-score is a measure of the standard deviations of an element from the mean of the distribution. The more positive the z-score, the worse the metric was relative to the average seen in the sample. More positive differential values will be those where the z-score for $S^?$ is larger than that of S^0. For metrics where $sgn = +1$ worse is a more negative value and the z-score calculation is inverted. As a result, smaller and more negative z-scores for $S^?$ over S^0 will result in larger differential values.

A z-score is a count of how many standard deviations a value is above the mean within a distribution. A visualization of a normal distribution with a mean μ and z-score standard deviation σ steps is shown in Figure 5.5.

One of the biggest benefits of using a z-score style calculation is that it can normalize the values of metrics measured with differing scales. Instead of comparing values directly, it can compare the deviations relative to the mean of the sample. This is also useful for assigning the weights for the summation of the effectiveness measure *eff*. The subsequent normalization allows the weightings to be used relatively. For example, a weight of 2.0 is approximately as influential on the search process as a

value of 1.0, and 0.5 half as influential.

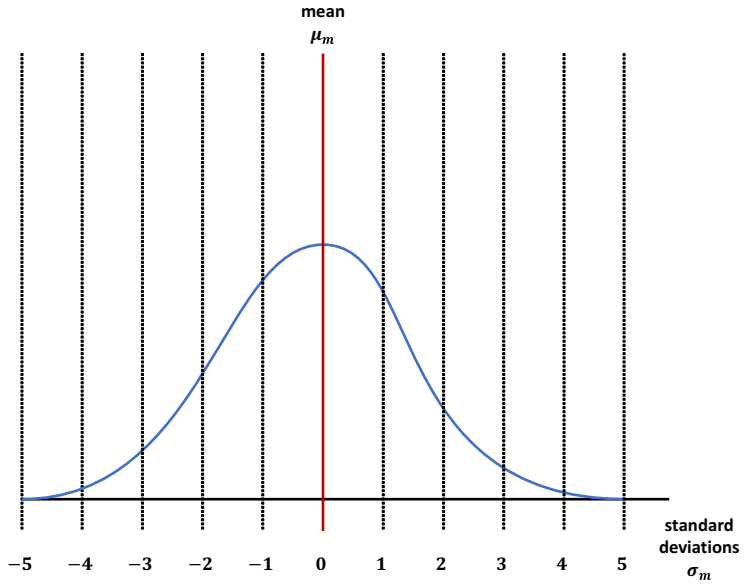


Figure 5.5: Z-Score in a Distribution

It is important to note that concept of z-score is being used here as a tool to enable the normalization of metrics for summation, not as a rigorous statistical measure. Some metrics will conform to being independent and identically distributed random variables, meaning that a sample distribution drawn from them can be approximated rather accurately as a normal distribution due to the Central Limit Theorem [NIS11]. However, for a number of metrics a normal distribution is not as accurate of an approximation. As will become evident in the experiments in Chapter 8, there will be metrics with long tails, truncated tails, or bi-modal distributions. There is statistical inaccuracy when approximating these distributions via a normal distribution. Despite this statistical inaccuracy, the z-score method of normalization is still a simple, yet powerful, utilitarian way to attempt to tackle the problem of disparate orders of magnitude and ranges between metrics such that they can be combined within the summation of the fitness function.

5.3.1 ESiD Example (continued)

This final subsection completes the example of the previous two sections with a simplified description of a fitness function evaluation for an individual in the evolutionary search.

The fitness function fit^{ESiD} evaluates each individual indi^{ESiD} after execution of the associated event sequence *es* to determine a fitness value. This value is based on *perf* and *eff* which each are based on the set M of metrics.

For this example, there will be one functionality metric m^{succ} that indicates if an event sequence successfully completed execution, and one performance metric m^{time}, which is a measure of the time for the event sequence execution to complete. As a result, $M = \{m^{succ}, m^{time}\}$

For the success metric $m^{succ} = ([1,1], N/A)$, success is measured as a value of 1. However, a measured value of 0 indicates failure, since it is outside the acceptable range $[1,1]$. Being unable to complete a desired sequence of interactions is an example of a failure value of 0.

For the time metric, $m^{time} = ([0,+\infty], -1)$. The range $[0,+\infty]$ means any time is functionally acceptable. Based on $sgn = -1$, better performance is to minimize this metric towards 0.

In the fitness function, *perf* will use m^{succ}, while *eff* will use m^{time}. The fitness function will use a weight $\mathit{weight}_{perf} = 100$ and a $\mathit{weight}_{eff} = 1$.

If m^{succ} fails (returns 0), then *perf* will evaluate to 0 and $(1 - \mathit{perf}) * \mathit{weight}_{perf} = 1 * 100 = 100$. This value is designed to be greater than the summations expected from the effectiveness measure *eff*. However, if the m^{succ} passes, then the fitness function relies only upon the value returned from the effectiveness measure *eff*. If there is concern that functionality failures may mislead the search, then the weight can be adjusted to be $\mathit{weight}_{perf} = 0$.

The effectiveness measure $\mathit{eff}(S^0, S^?, M, es)$ is a summation over the metrics in M that are performance metrics (m^{time} in this example). These metrics are measured for the event sequence es for both the baseline system S^0 without the security mechanisms and the system $S^?$ with the anti-virus security mechanisms installed on it. The weight for the one metric m^{time} will be $w_{time} = 1$.

The effectiveness measure $\mathit{diff}(v(es), v^{base}(es))$ relies on comparing the observed metric values for m^{time} during the execution of the event sequence es. The observed value $v(es)$ is from the system $S^?$ with the anti-virus security mechanisms installed on it, and the observed value $v^{base}(es)$ is from the baseline system S^0 without the security mechanisms.

These observed values are compared relative to the distribution of the values as previously seen across a sample of random event sequences of the same length to get a z-score. This z-score is a numerical value of how many standard deviations a value is above/below the mean of the distribution. The secured system $S^?$ will be considered to have a sample mean $\mu = 10$ and a sample standard deviation $\sigma = 2$. The baseline system S^0 on the other hand has a mean $\mu^{base} = 5$ and a standard deviation $\sigma^{mean} = 1$.

For one evaluation of the fitness function for event sequence es, the fitness function evaluates $v(es) = 14$ and $v^{base}(es) = 6$ for the metrics m^{time}.

As a result, the z-score for $S^?$ is:

$$\frac{v(es) - \mu}{\sigma} = \frac{14 - 10}{2} = 2$$

The z-score for S^0 is:

$$\frac{v^{base}(es) - \mu^{base}}{\sigma^{base}} = \frac{6 - 5}{1} = 1$$

The differential of z-scores is $2 - 1 = 1$. The $sgn^{time} = -1$ tells us that the metric

m^{time} is better when it is minimized. Larger values are less desired. The differential function *diff* applies the directional adjustment of $-sgn^{time} = -(-1) = 1$ and returns the value of 1. This value can be read as the z-score of the metric m^{time} is 1 standard deviation worse for the secured system than it was for the baseline system.

The fitness function multiplies the returned differential value by its weighting w^{time} and ensures that the result is a greater than zero using the Max function. These differential values are then accumulated via summation to form the effectiveness measure *eff* for the fitness fit^{ESiD} of the event sequence es representing the individual $indi^{ESiD}$.

Chapter 6

Software Implementation

This chapter describes an implementation of the previously described abstract genetic algorithm as an exploratory search tool. The purpose of this tool is to allow its operators to examine the performance and operation of different combinations of security mechanisms. The tool's exploratory search process evaluates interaction event sequences as scripts executed on virtual machine instances. These virtual machine instances are configured with both the programs to enable the event sequences' interactions and the security mechanisms being examined.

The exploratory search tool converts event sequence combinations into program execution scripts that are then executed on virtual machine system configurations. The resulting metrics are extracted and used to evaluate an event sequence's fitness. The fitness values of the individuals in the current population are used to drive the creation of future event sequence combinations to explore. Event sequence combinations identified by the search algorithm as being of interest are reported. These reported event sequence combinations can be examined in greater detail by using the infrastructure of the tool to automate its repeated execution and evaluation. Types of examples produced by the tool include where system operation experiences unacceptable delays, and where the system is unable to complete a sequence of interactions.

This chapter consists of four parts. First, Section 6.1 describes the structure of the tool and how the genetic algorithm search process is implemented. Next, Section 6.2 discusses the configuration of the virtual machine instances. Section 6.3 elaborates on the conversion process of an event sequence into a script that can be executed on a virtual machine instance. Finally, Section 6.4 describes an example implementation

framework and an example of a script produced for that framework.

6.1 Tool Structure

The black box view of the tool structure is uncomplicated. The user indicates the parameters of a search experiment via a configuration file and begins the search process. The search process executes until the indicated number of generations is reached, or until otherwise interrupted. During this execution, the tool compiles and updates output log files containing information about the exploration accomplished during the search experiment.

Internally, the black box of the tool is more complicated. Within the tool, there are dependencies between the configuration file and both the virtual machine instances and code implementation. For example, enabling experimental configurations that involve event sequences that include interactions for a specific program requires a couple steps to be taken. First, that program needs to be configured on the virtual machine instances. Second, code must be created to represent and convert that program's event interactions into executable script actions for the virtual machine.

Similarly, there are some broader structural configurations that must be completed to enable an experiment to take place. The chosen operating system environment and security mechanisms being tested must be installed and configured on the virtual machine instances. Additionally, and independent of a specific experiment, the virtual machine manager utilized dictates the API used to communicate between the code-base and virtual machine instances. The internal tool communication between the exploratory search process and the virtual machine instances must be developed around the structure and limitations of this virtual machine manager API.

Most of the parameters for the experimental configuration file are numerical. However, some can be considered structural as they indicate the inclusion of different

programs and/or associated interactions in the event sequences explored in the experiment. These structural parameters require structural representation within the code-base. The first structural configuration is that of the virtual machine instances to include the indicated programs. This structural configuration is discussed in Section 6.2. The second type is the creation of the methods in the system's code to represent and convert interaction events into script commands for the virtual machine. This event sequence to script conversion is described in Section 6.3.

The basic external structure of the tool can be seen in Figure 6.1.

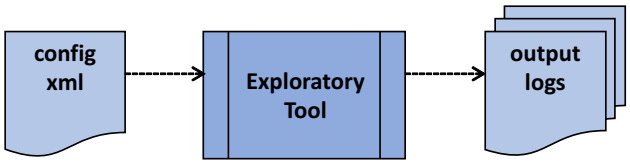


Figure 6.1: Basic External Structure

The types of parameters for a search experiment include five main categories: experiment, program, virtual machine, genetic algorithm, and fitness function parameters.

1. General experiment parameters include the length of the event sequences for the search, and how many results to include in the output log listings.

2. Program related parameters contain what programs make up the search space, and what configuration values do they need.

3. Virtual machine configuration parameters include where the virtual machine manager is installed, what virtual machine instance and snapshots to use, and what parameters related to usernames/passwords/installed programs are needed.

4. Genetic algorithm specific parameters include how many generations to execute, population size, what genetic operators to use, and how often to use different operators.

5. Fitness function parameters primarily consist of weighting numbers for the genetic algorithm's fitness function. In effect, which metrics to include in the fitness and the relative weighting between them.

The output log files have two primary purposes:

1. The first purpose is to track the history of the search process. In particular, that purpose is to track the top individual of each generation and the metric values associated with it. In typical circumstances, the top individual or individuals at the end of the search process are the primary candidates for examination by the user.

2. The second purpose is to track outliers during the search process. The search is directed by a median fitness designed to reduce the impact of host machine scheduling or other events external to the virtual machine from biasing the search too far towards one single evaluation of an individual. However, outliers are also possible examples of extreme forms of inefficient performance or failed functionality. As a result, outliers are tracked for closer examination.

Within the exploratory tool itself there are two halves. On one side is the system core, which consists of the compiled code of the tool. On the other side is the virtual machine manager. This representation can be seen in Figure 6.2.

The system core executes the exploratory search genetic algorithm. When evaluating an individual in the fitness function, the system core translates an individual event sequence representation into a program execution script. Next, it requests this script to be run on a baseline snapshot of the virtual machine instance, and then on a snapshot with security mechanisms installed. Metrics are requested from the virtual machine manager for each of these executions. The extracted and processed metrics are then interpreted within the fitness function evaluation.

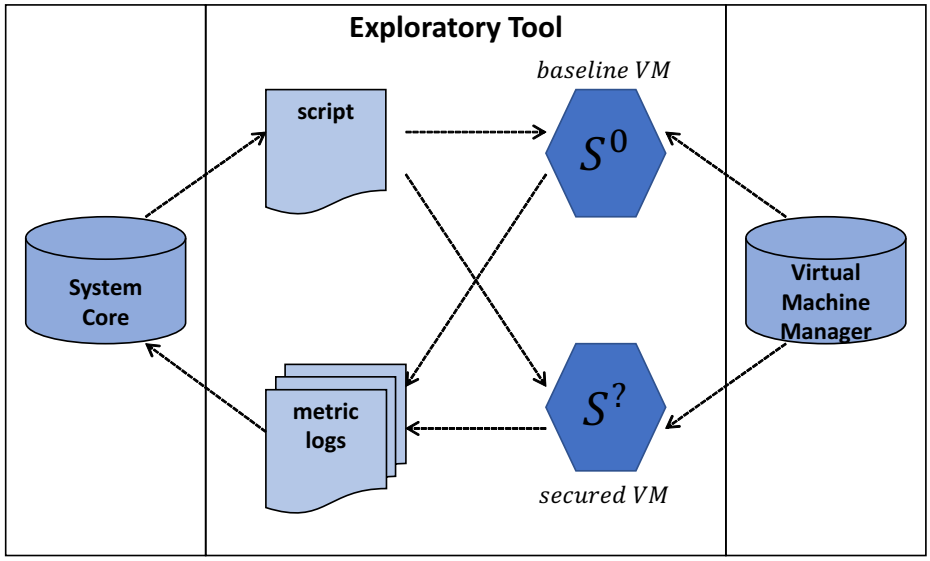


Figure 6.2: Internal Tool Structure

The base functionality of the tool is to perform the genetic algorithm exploratory search experiments. These experiments produce a set of prospective event sequences for further examination. One of the steps to examining these prospective event sequences is compiling evaluative data on the repeatability of the performance and functionality results of the system during these event sequences. To accomplish this, the exploratory tool is capable of a second mode of operation that makes use of the same infrastructure used for executing the exploratory search. This second mode performs multiple successive simulations of a single event sequence. During these repetitions the metrics of performance are tracked to provide data for the event sequence's verification and investigation. The use of this secondary mode's successive simulations is one of the processes described in Chapter 7.

The search process of the genetic algorithm of the exploratory tool is depicted in Figure 6.3.

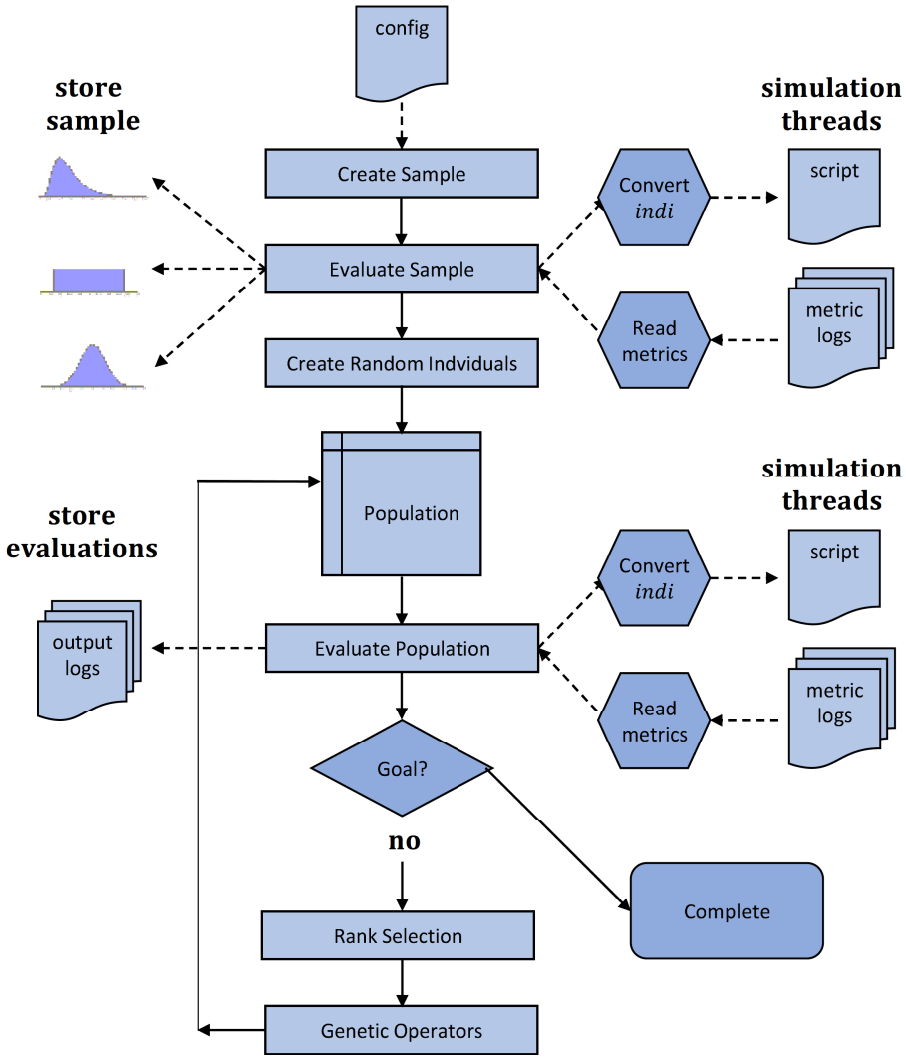


Figure 6.3: Search Execution Structure

The steps of the search process are as follows:

1. Parse configuration parameters from the configuration file.

2. Create a representative sample consisting of event sequence individuals.

3. Evaluate each individual in this sample.

   - This evaluation requires the individuals to be converted into a script representation and then for the resultant script to be executed on the baseline and secured virtual machine instances.
   - The metrics from the event sequence's execution are requested from the virtual machine manager and stored for each metric.

4. The result is a sample distribution of performance for each metric.

5. Create an initial population *pop* of random event sequence individuals.

6. Start the genetic algorithm loop and continue within it until termination is requested, or a goal such as a specific fitness value or other event sequence/population characteristics is met.

   (a) Evaluate each individual in the population.

   - This evaluation follows that of Step 3 of the sample creation.
   - However, instead of storing the metric values, the metric values are used by the fitness function to calculate a value based on their relationship to the stored sample distribution values.
   - This fitness value is stored in the individual.
   - As individuals are evaluated, output logs are updated to track the history of the search, outliers during the search, and the top individuals found.

(b) If a termination goal is reached, then the genetic algorithm exits its loop and completes the search process.

(c) If not, then rank-based selection is used to select individuals from the population.

(d) Rank selected individuals are processed by different genetic operators to produce new individuals for the next population (generation) of the search algorithm. The rate at which different genetic operators are chosen is based on the configuration parameters.

(e) Loop to evaluate the newly created population.

7. Search process is complete.

The evaluation of a single event sequence requires the existence of a virtual machine instance capable of representing the system state of the baseline system and the secured system with the security mechanisms installed. The structural configuration of the virtual machine instances is discussed in the following section. The conversion process from an event sequence to a script is discussed in the section following that.

Multiple copies of the same virtual machine instance can be used to parallelize the evaluation of the initial sample set of individuals and each population evaluation during the search process. Each evaluation workflow is an independent process that can be threaded. The general limitation is the number of parallel virtual machine instances that can operate simultaneously without disrupting each other on the testing system. The search process workflow centralizes information from the evaluation phase during the rank selection and genetic operator steps to create the next population. As a result, the upper limit on parallelization gain is the number of individuals in the population. If desired, this parallelization could be distributed amongst multiple networked machines by passing the desired scripts to client machines and returning the resultant metric logs to the central server. Since the execution of an

event sequence script has an order of magnitude of seconds to minutes, the network delays associated with this type of solution are not a limiting factor.

There are some considerations to be made when deciding on parallelization. The virtual machine management process on the host machine occupies host system resources. Depending on the environment chosen, these loads may cause a large variability in measured statistics during the evaluation of an event sequence. At a certain point, these host system variations may overwhelm the ability of the fitness function to differentiate individuals by the composition of their events. There are similar dangers with network parallelization. In order to share the same sample metric distribution, the networked machines must be homogenous to prevent the differences in host machines being the cause of different fitness evaluations.

6.2 Virtual Machine Instances

Virtual machine instances are used to simulate the execution of the event sequence individuals that make up the population of the exploratory search. For each search experiment, it is necessary to configure two virtual machine instance templates. The first is for the baseline unsecured system. The second consists of the baseline template supplemented with the installation and configuration of the security mechanism(s) being explored. This second template is the secured system. These templates can be reused across multiple experiments that share the same operating system, installed programs, security mechanisms, and associated configurations. These templates may be instantiated multiple times to allow for the parallelization of the evaluation step of an individual.

Virtual machine instances necessarily exist within some virtual machine manager infrastructure that enables their creation and execution. The chosen virtual machine manager should also have the following features.

1. A virtual machine manager that allows for automated commands to launch and terminate virtual machines should be chosen. It may be necessary to terminate virtual machines that freeze/crash and can no longer be shut down gracefully.

2. The virtual machine manager should allow for execution of programs installed on the virtual machines. Particularly, the capability to initiate the script execution that represents an event sequence individual.

3. The virtual machine manager should allow for operational metrics to be extracted when requested. This can either be completed via API requests or by file access to information output during the event sequence's script execution.

The choice of virtual machine manager will dictate the code interface within the core of the tool that interacts with the virtual machine instances. This code interface utilizes the virtual machine API to interact with the available virtual machine management commands. If parallelization utilizes multiple networked operating environments, then this code interface would need to be supplemented with a network layer of communication.

The first step to configuring a virtual machine instance is the determination of the operating system. This choice dictates the external and virtual hardware necessary for event sequence simulation and hardware emulation. The choice of operating system then dictates the available programs and security mechanisms that are compatible with the chosen operating system. The parameters of emulated hardware should also match the requirements of the user. The most important of these are processor capability, disk space, and network capabilities. With these settings determined, the initial template of the virtual machine instance can be created.

It is recommended that a snapshot of this initial virtual machine instance state be maintained, since setting it up is a time consuming yet fundamental step. From this initial fresh installation state, multiple branches of further installation can be

followed. However, there is generally a common sequence of initial virtual machine configuration steps applied to this fresh installation state. This includes ensuring proper hardware installation, disabling annoyances such as sound output, and configuring human-computer interaction related settings such as resolution. Additional steps often include disabling scheduled events such as update processes or pop-up events that could interrupt future operation. A second snapshot is recommended after this configuration.

The four fundamental configuration areas required of the virtual machine instance are depicted in Figure 6.4.

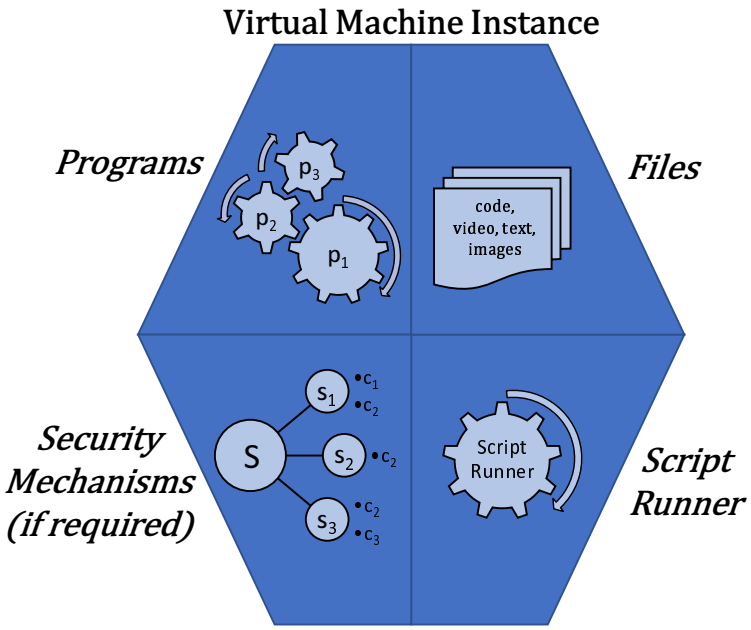


Figure 6.4: Virtual Machine Structure

The second snapshot often contains the installation and configuration of the script running program. This is because the script running program is dictated by the exploratory tool's code implementation, and remains static across all experiments created. The other three configuration areas are dependent on the experiment being completed. The chosen programs of the experiment and the files to enable their inter-

actions are dictated by the experiment's parameters. The installation and creation of these two configuration areas completes the baseline unsecured system. A snapshot must be taken at this point so that this system state can be used for the first half of each interaction event sequence evaluation.

The security mechanisms are the last to be installed. After their addition to the baseline snapshot, the result is the secured system snapshot. This system state can now be used for the second half of each interaction event sequence evaluation. The previously installed programs and files exist to populate the system environment, in which the desired interactions of the event sequences are executed by the script runner.

The result of this creation process is generally a virtual machine with 4 snapshots: a fresh operating system snapshot, an initial setup and script runner installation snapshot, a baseline system snapshot, and a secured system snapshot. This virtual machine can then be duplicated as necessary to enable parallel execution. Different experiments sharing a similar baseline can be created by installing a new configuration of security mechanisms to create a new secured system snapshot.

The design of the exploratory search tool is greatly influenced by the choice of script running program. Like the choice of the virtual machine manager, the choice of script runner dictates the API available to trigger automated program interaction. In particular, the script runner dictates how programs can be started, stopped, and their interactions automated. Interaction with the script runner is defined by a scripting language. Each program interaction will have to be encoded into this scripting language. This process is discussed in the next section of this chapter.

6.3 Event Sequence to Script

The conversion process of an interaction event sequence individual into a script consists of two primary parts. For the first part, each program necessarily requires a representation within the system code. This representation manages the configuration of the program from the experiment parameters as well as dictating the available interactions and their associated script representation. The second part is a conversion algorithm that receives an interaction event sequence individual from the exploratory search process as input, and produces a script as output. This script is in the script runner's language and can be stored in a file accessible by the virtual machine instance. Execution of the script file is requested through the virtual machine management API.

The representation of a program p within the system code consists of the following parts.

- Parsing and storing the parameters from the experiment configuration file to configure the available options of the program and its interactions.
- Script statements necessary to import script language APIs particular to enable interaction with that program ($import_p$).
- Script statements to create variables that enable the program interactions. Often these variables are configured by the previously parsed parameters. An example is the locations of files that are used with by the program ($vars_p$).
- Script statements to start/close or make the program active. ($active_p$)
- Finally, script statements for each interaction that can be used for that program. Determining which of these interactions to use can be indicated in the experiment program parameters parsed earlier (k^{th} interaction of program p is $interaction_{p,k}$).

This program representation is the reference material that the conversion algorithm uses to create a script from an interaction event sequence. The import and variable script statements are collected to form the start of the produced script, while the start/close/active statements are inserted into the sequence of interactions as the experiment parameters dictate. Some experiments will dictate that all programs are started at the beginning of the script, or when the first program interaction is taken. Other experiments will close programs after their final interaction or leave all programs open until the script completes. The statements to make programs active are inserted whenever the sequence changes from one program to another. This ensures that interactions that require focus are not disrupted. Due to timing delays, the script statements to make a program active are often looped until the active state requested is achieved.

If there are metrics that relate to a specific program interaction that must be measured during script operation, then these script statements should be inserted during the script conversion process. In general, this process would consist of wrapping a program interaction, or sequence of interactions, with a measurement script operation. This wrapper would begin with an initial script statement that begins metric calculation, such as a timestamp, followed by a final script statement that finishes the metric calculation. This metric calculation should then be pushed to an output metric file that will be read by the exploratory search tool's fitness evaluation process.

The general case of this type of metric is script duration. This duration is measured by a script statement that takes a timestamp at the start of the script and a final script statement that executes at the end of the script to store the elapsed time to an output file.

One unique experiment type consists of an exploratory search for sequences of interaction events that are preceded/followed by an existing static sequence of events.

The goal of this type of experiment is to use a set of interaction events with known or expected behaviour to influence the search process. In this case, before the interaction event sequence is processed by the conversion, this event sequence prefix/suffix of static interaction events is added to the sequence being converted.

The general interaction sequence to script conversion algorithm is as follows:

1. Add the optional prefix/suffix interaction events to the event sequence.

$$es = es_{prefix}, es_{initial}, es_{suffix}$$

2. Determine all the programs P_{es} used in this event sequence es.

3. Add script comments that indicate the sequence being output for debugging/tracking.

4. For each program $p \in P_{es}$, output the script import statements $import_p$.

5. Add script statements $start_{elapsed}$ to initialize script duration tracking.

6. For each program $p \in P_{es}$, output the script variable statements $vars_p$.

7. If programs are all started at beginning of script, then for each program $p \in P_{es}$, output script program start statement $start_p$.

8. For each interaction event $ip = (j, k) \in es$ where j indicates a program $p_j \in P_{es}$:

   (a) If interaction's program is started immediately before this interaction, then output script program start statement $start_p$.

   (b) Output script statement $active_p$ that makes the program active.

   (c) Output interaction event script statement $interaction_{p_j,k}$.

   (d) If interaction's program is ended immediately after this interaction, then output script program end statement end_p.

9. If programs are all ended at end of script, then for each program $p \in P_{es}$, output script program end statement end_p.

10. Add script statements $end_{elapsed}$ to finalize script duration tracking and output this value to an output file.

If the evaluation of interaction event sequences is parallelized, then it will be necessary to configure this script creation process such that the name of the output file for each script is unique to the virtual machine instance upon which it is executed. Additionally, the output file for the script duration (or internal interaction event metrics) will also have to be unique. Similarly, the output file when virtual machine metrics are requested from the virtual machine instances will have to follow the same practice. This unique naming will prevent files from overwriting each other.

Another important step for the management of the script files and output files is the deletion of these temporary files after the completion of the evaluation of each interaction event sequence. The execution of event sequences contains the possibility that the execution of the script may be incomplete, or the virtual machine operation be terminated forcefully. It is undesirable for the input/output from a previous event sequence to be accidentally misused by the current event sequence. The deletion of these files after their use and before a new event sequence evaluation begins is a procedural step that ensures information consistency.

6.4 Example Implementation

This section describes an example implementation of the abstract structure described previously. The level of detail of this section attempts to be detailed enough to establish an understanding of the required structure and challenges addressed to implement the exploratory search via event sequences methodology for the problem

of Effective Security-in-Depth, without requiring unnecessary specific knowledge of the implementation choices.

For the example implementation, it is necessary to first describe the environment of the experiment that is going to take place. The choice of operating system in this example is an installation of Microsoft's Windows XP Service Pack 3 (WinXP) installation. This operating system needs to be installed into a virtual machine managed environment. This chosen virtual machine management tool is Oracle's Virtual Box version 5 (VBox). A set of experimental programs also needs to be chosen. This example will be using Microsoft Word/Excel for document operations, VideoLan VLC Media player for video and audio playing, and Java JDK for java program compilation. Lastly, AutoIt will be used as the windows automation and scripting language.

To setup the environment of the virtual machine instances required for an experiment, WinXP must be installed into a virtual machine. The chosen virtual machine should have enough memory for the operating system to operate normally: for example, 2 gigabytes (GB) of RAM and 20 GB of hard drive space. WinXP can then be installed into a fresh installation state. This installation state should be saved with a snapshot.

Then the machine can be configured into the initial desired state. This configuration involves dealing with new installation notifications, configuring the basic resolution, dealing with Windows registration notices, and setting up the virtual machine to see the host machine's files. Another step found necessary is disabling the time advancement of the virtual machine so that running tests are not influenced as the local machine's time advances. Finally, AutoIT is installed to enable script execution on the virtual machine. This secondary fresh installation state should also be saved as a snapshot.

Next, the virtual machine needs to be set up into the un-secured (**Without**) virtual machine instance state. This state requires the files needed by the experimental set of programs to be transferred into the virtual machine. For this example, it was necessary to copy over a set of files for document operations, video operations, and compilation operations related to the programs involved in the experiment. Then each program is installed and verified to ensure that they launch correctly. With this complete, a snapshot is made for the **Without** virtual machine system state.

Finally, the secured (**With**) virtual machine instance state is configured. This state only requires us to install the security mechanism, or mechanisms, being tested. The specific chosen security mechanism is not important to this example, but in general it can be assumed that a standard anti-virus program was installed, such as ClamWin, Norton, Avast, AdAware, etc. With this complete, a final snapshot is made for the **With** virtual machine system state.

If this system setup has been configured before for a previous experiment, then it may be possible to move on to running a similar experiment. However, for a new experiment there are a number of steps that have to be performed.

1. The first step is configuring the exploratory tool to launch the virtual machine, to close the virtual machine either gracefully or ungracefully, and to execute instructions to run the desired AutoIT script on the virtual machine instance. This requires a framework within the tool that understands the VBox API structure. This then requires troubleshooting system specifics that deal with the execution and management of VBox operations. Terminating an unresponsive virtual machine is an important challenge. For the metrics of each script execution to be relatively consistent, the virtual machine must start the execution of each script from roughly an equivalent system point. A crashed virtual machine needs to be recovered and restored such that the simulation of the next script can start as

expected.

2. The second step is configuring the tool to create AutoIT script code that will launch program actions for each of the installed programs. Each of these actions needs to correctly interact with the files installed. This requires creating software structures for each program that manage configuration parameters in the tool. With these structures, it is then necessary to create operations to translate each action into script code. These actions need to be designed so that they can be accumulated into more complete scripts that run on the virtual machine. The actions and their combinations then need to be tested to ensure they operate as desired and are created consistently.

3. The third step is getting output from the virtual machine manager. The tool must be configured to parse script created files that contain statistics about script performance. An example of this is a file that the script creates to contain the execution time of the script. This length of time is separate from the execution time of the virtual machine and only calculated from when the virtual machine is operational and the script begins execution. This script-produced time avoids the inconsistencies in the time required when waiting to start and stop the virtual machine instances. The tool also needs to interact with the virtual machine manager API to request its virtual machine instance metrics and statistics, which include processor, hard drive, memory, and network measurements.

Much of the software development required for each of these three steps is contained within the scope of the fitness function. The rest of the exploratory tool is implemented close to the abstract methodology described in Section 5. The general search process only needs to be aware of the name and number of programs that exist and how many actions exist for each program. The configuration responsibilities for programs and their interactions can be delegated to the program specific code. It is

quite possible to replace the implementation of the fitness function's virtual machine interaction with that of some other virtual machine management software without changing the search process code.

The fitness function itself is responsible for translating the abstract program actions of an individual into a script. It then executes this script on each of the **With** and **Without** virtual machines. The fitness function requests the metrics from these two execution attempts to create a fitness value along with other evaluation information. This information is then stored in the individual.

An example of the format of a configuration file for an experiment can be seen in Figure 6.5.

This configuration file is pushed into the tool. From the configuration file, the tool knows how many programs are in the experiment. From the parameters given to each *program*, the tool knows exactly how many actions exist for each program. This information allows the tool to create the indicated *sample_size* number of sample individuals for the experiment. The length of the event sequence within each individual is indicated by the *dimensions* parameter.

The tool evaluates each individual in the sample. The metric values from this evaluation are accumulated to create a sample mean and standard deviation for each of the metrics listed in the weight manager. The tool performs this evaluation by using the indicated fitness function *fit*. The fitness function executes an individual on the indicated virtual machine *vm* using the virtual machine instances *with_instance* and *without_instance*.

Next, the tool progresses into the regular genetic algorithm search process. The tool makes an initial random population of size *pop_size*, and evaluates these individuals again using the fitness function parameters. Now instead of sample metric values, the fitness function stores a fitness value created by judging each produced

```
<Config>
  <Name>Experiment</Name>
  <Seed>rng_seed</Seed>
  <Dimensions>dimensions</Dimensions>
  <SampleSize>sample_size</SampleSize>
  <Top>top</Top>
  <FitnessFunction>
    <Name>fit</Name>
    <VM>vm</VM>
    <Without>without_instance</Without>
    <With>with_instance</With>
    <Parameters/>
  </FitnessFunction>
  <Experiment>
    <PopulationSize>pop_size</PopulationSize>
    <Generations>generations</Generations>
    <MetricWeightManager>
      <Weight name="metric">weight</Weight>
    </MetricWeightManager>
    <SelectionProcess>sel_proc</SelectionProcess>
    <Operators>
      <Operator>
        <Name>mut_op</Name> <Weight>weight</Weight>
      </Operator>
      <Operator>
        <Name>cross_op</Name> <Weight>weight</Weight>
      </Operator>
    </Operators>
  </Experiment>
  <Progams>
    <Progam> <Name>progam</Name> <Parameters/> </Progam>
  </Progams>
</Config>
```

Figure 6.5: Example Experiment Configuration

metric value relative to the sample metric distributions and weighted by the weights given in the weight manager.

For each subsequent generation's population, new individuals are formed using the indicated operators. These operators create new individuals using individuals selected from the current population via the selection process *sel_proc*. The genetic algorithm process completes after *generations* number of iterations have passed.

During this operation, the *top* number of individuals found over the search will be stored and printed out to log files. Other unique results, such as individuals that crashed the virtual machine or that did not finish script execution, will also be tracked in log files. These logs files are then explored during and after the tool's execution to determine what was found during the exploratory search.

Interesting results can then be re-examined using the tool with a similar configuration file. Instead of being an *Experiment*, that block is replaced by a *Test* block of configuration instructions. These instructions indicate a specific *script* stored in a file accessible to the virtual machine. This script is then executed *count* number of times, and the metrics related to this operation tracked in log files.

An example test configuration can be seen in Figure 6.6.

Each individual in the search process has an event sequence that must be converted into an AutoIT script when being evaluated on the virtual machine instances. Each program that appears in an event sequence has pre-amble requirements that must be included in any script to enable AutoIT to execute individual actions. These include necessary libraries and configuration variables for its actions. These variables will generally include data or locations of the data stored on the virtual machine. Examples include video files, code files, and documents. If necessary, each program also needs to know how to start itself such that actions can then be run for it. Lastly, a script is wrapped by a timer that writes to a file for this example experiment. This

```
<Config>
  <Name>Experiment</Name>
  <Seed>rng_seed</Seed>
  <Dimensions>dimensions</Dimensions>
  <SampleSize>sample_size</SampleSize>
  <Top>top</Top>
  <FitnessFunction>
    <Name>fit</Name>
    <VM>vm</VM>
    <Without>without_instance</Without>
    <With>with_instance</With>
    <Parameters/>
  </FitnessFunction>
  <Test>
    <Script>script</Script>
    <Count>count</Count>
  </Test>
  <Progams>
    <Progam>
      <Name>progam</Name>
      <Parameters/>
    </Progam>
  </Progams>
</Config>
```

Figure 6.6: Example Test Configuration

timer indicates how long the script took to execute.

The following is the example of a form of an individual that may exist in the example:

$[.Empty\text{-}0, .Word\text{-}8, .VLC\text{-}0, .Compile\text{-}1, .Excel\text{-}4]$

This example individual will be converted into the script in Figure 6.7 on the following page.

This chapter described the search tool implementation of the exploratory search process described in Chapter 5. The following chapter describes a selection of experimental processes using this tool.

```
#include <MsgBoxConstants.au3>
#include <Word.au3>
#include <Excel.au3>
$start = TimerInit()
;Data
;CODE FOR PARAMETERS FOR ACTIONS

;Program class programs.Empty action 0
Sleep(1000)
;Start program class programs.Word
Local $vWord = _Word_Create()
Local $vWordDoc = _Word_DocOpen($vWord, $cWordFilePath)
;Program class programs.Word action 8
Local $vWordRange = _Word_DocRangeSet($vWordDoc, -1)
_Word_DocTableRead($vWordDoc, 1, 1)
;Start program class programs.VLC
;Program class programs.VLC action 0
Run($cVLC)
;Program class programs.Compile action 1
RunWait("C:\VMs\Store\Compile\remove.bat", "C:\VMs\Store\Compile")
;Start program class programs.Excel
Local $vExcel = _Excel_Open()
Local $vExcelWorkBook = _Excel_BookOpen($vExcel, $cExcelFilePath)
;Program class programs.Excel action 4
Local $vExcelRange = $vExcel.Activesheet.Range("B1")
_Excel_PictureAdd($vExcelWorkBook, Default, $cExcelImage, $vExcelRange)

$diff = TimerDiff($start)
$timerfile = FileOpen("\\VBOXSVR\Files\timer.dat", 1)
FileWrite ( $timerfile, $diff )
FileClose ( $timerfile )
```

Figure 6.7: Example AutoIT Script

Chapter 7

Exploratory Search Processes

This chapter describes processes built around using the exploratory search tool, introduced in the previous chapter, to produce solutions to the Effective Security-In-Depth problem. Rather than approaching these processes directly from the abstract formalization given previously, these processes are described from the point of view of the user engaging in their use. Each process is first framed by the process goal, and then the steps designed to achieve it are introduced.

To begin, Section 7.1 discusses the initial process of filtering the pool of available security mechanisms under consideration. Next, Section 7.2 introduces the process of a user attempting to choose a singular security mechanism given their usage profile. This is followed, in Section 7.3, by the process of a user attempting to choose between combinations of security mechanisms.

Each of these processes makes use of two key stages. The first stage is the creation and execution of experiments using the tool. Each of these experiments is an evolutionary search for interaction event sequences. The result of any experiment is a collection of logged event sequences from individuals that were either examples of extreme fitness values, or examples of emergent misbehaviour such as functionality failure. The second stage is to quantitatively assess these examples in order to make recommendations and conclusions based on them. This second stage generally includes utilizing the tool in its secondary role to verify the performance of an event sequence example.

Shared between the two processes described in this chapter is the challenge of verifying the most extreme examples, such as functionality failure, where an event

sequence fails to complete due to conflict between programs or virtual machine failure. Examples of these are tracked in the log files of previous experiments but cannot be directly compared to event sequences that completed and have a fitness value. A final process, described in Section 7.4, is designed to verify and assess whether or not one of these candidate examples is of interest.

7.1 Initial Security Mechanism Filter

It is in the best interest of individual users to protect their computer from undesirable malicious interactions. This requires the installation of one or more security mechanisms. However, as discussed previously, security mechanisms are generally not designed to inter-operate. In fact, many security mechanisms will identify the installation of other security mechanisms and advise against, or even prevent, simultaneous installation. The great majority of users also do not have the skillset and desire to individually test combinations of security mechanisms. Therefore, the most common circumstance is that a user wants to select between different individual security mechanism choices. However, the most desirable circumstance would be for the user to select a combination of multiple security mechanisms that provide security-in-depth.

Users generally rely on a combination of three sources of information when choosing a security mechanism. These three sources are: what other users have told them, what the security mechanism creators claim on their advertisement material, and what review material recommends.

1. User reviews appear helpful on the surface but are full of unstated biases. User reviews are generally bi-modal between groups of users who had extreme reactions to the product. These types of selection biases in user reviews have an impact on the value of the reviews [HZP09]. There is a lot of uncertainty in relying on the reviews of a user who very likely has a different usage profile and system

configuration.

2. Security mechanism creators know their product the best, but their goal is primarily to convince the user to purchase the product. The best a user can expect is that some effort will be made to create a good enough product to encourage a secondary purchase in the future. Many products come bundled on computing systems. In addition to being a marketing tactic, this bundling is also a subtle admission that most users may not search out the product or given the opportunity would pick another instead.

3. Review materials are often the best source of information, but must be approached with caution. Less ethical review sites will have been swayed by free software provided by some of the product creators. More trustworthy review sites will have obtained their security mechanisms without relying on free access. The most common review setup is a reviewer installing the mechanism on their own machine and then providing feedback, possibly with a couple of scripted event sequences against which performance is benchmarked. The result is generally better than users' reviews. This improvement is due to the addition of a more expert opinion and quantitative consideration.

   The best review sites will expand on the benchmark concept with usage benchmarks for a couple different types of usage. However, the time taken to produce these type of benchmark suites means such review sites are rare and that the reviews are completed less often. Even in the case of the best review site, if a user has a different system configuration and a usage profile not explored, then the review will have limited value. In fact, the process methodology proposed in the thesis was designed because of the possible limited value of these static benchmark suites.

These three sources of information are still useful to the user. They should allow a user to reduce the initial number of security mechanisms being considered to fewer that have the security coverage features they want, fall within the costs they are willing to pay, and pass muster for general purpose review benchmarks. Once the user has this more limited set of security mechanisms, they can make their final decision through the selection processes described below. These selection processes explore the dangers of each security mechanism, or security mechanism combination, within the context of the user's usage profile.

7.2 Process for Selecting a Single Security Mechanism

Goal: From a limited set of security mechanisms that provide the desired security coverage, select the security mechanism that provides the most robust coverage while limiting the performance cost. Perform this selection by judging performance cost based on how the installation of the mechanism negatively affects the performance of the system. This performance will be measured in regard to the user's usage profile, which defines the programs and interactions the user expects to use regularly.

Steps: Select a Single Security Mechanism

1. Collect the installation software for each of the security mechanisms.

2. Collect the installation software, and accompanying files, for each of the programs that form the user's usage profile.

3. Setup the experimental environment for the exploratory search tool. Create the virtual machine instances for the fresh installation state without security mechanisms. From this fresh installation, create one virtual machine instance for each security mechanism being compared. More detailed virtual machine setup and code configuration was described in the previous chapter.

4. Create and run an evolutionary search for interaction event sequences experiment for each of the security mechanisms.

5. Explore the search results to select representative event sequences produced by each experiment.

6. Validate the event sequences by evaluating them multiple times with the security mechanism that created them.

7. Evaluate the event sequences multiple times for each of the other security mechanisms.

8. Compare and contrast the quantitative differences in event sequence performance between the security mechanisms to select the one with the least performance cost.

7.2.1 Creating Experiments

There will be one experiment created in Step 4 for each security mechanism being compared. Each of these experiments will be identical except for the virtual machine instance $S^?$ used to represent the system state with the security mechanism installed.

An average user cannot be expected to be aware of, or care about, individual system statistics. The average user is concerned with how slow or inefficient the system appears to operate from their point of view. Therefore, the only fitness function component that will be measured in this example is the script completion time. A more advanced user may make use of experiments that include more components in the fitness function.

The programs chosen for the experiment are based on those that represent the user's usage profile. The genetic algorithm parameters are recommendations that should work for the majority of expected cases. The sample size chosen, in this case fifty, is chosen to be large enough that the user has reasonable confidence in the mean

and standard deviation of the distributions created for the fitness function metric of script completion time.

The most important remaining parameter choice for the user is that of the event sequence dimensions. The most suitable choice of dimension size is large enough that enough events can occur to cause interesting interactive behaviour, but small enough that too much time is not taken to evaluate the event sequences. For this example, the dimension size chosen is ten.

Figure 7.1 is an example of the configuration file for one of the experiments.

Once this experiment completes, the log files associated with the experiment will contain the top 100 event sequence examples for different categories of behaviours. The most commonly useful category is that of the top 100 examples based on their median fitness. However, other categories may include, top max fitness, top median time, top max time, most often incomplete, and most often crashed. The event sequences that occur in the most often incomplete or crashed categories are examined using the process in Section 7.4 of this chapter. It is common that event sequences in the categories tracking maximums are the result of simulation variance (sometimes due to host system behaviour) and in many cases are not reproducible. This was the reason that the median, instead of the maximum, was chosen for the fitness function to guide the search. The event sequence examples for cross-examination are generally chosen from the median fitness category, with possibly some selected from the maximum fitness. The top median/max time are of use for secondary examinations, especially when other fitness function metrics are used in addition to script time.

7.2.2 Quantifying Results

To make a selection decision, the user needs to be able to make quantitative decisions using the produced event sequence examples. During the exploratory search experiment, each event sequence example could have been simulated as few as one time, or

```
<Config>
  <Name>Single_Security_Mechanism_Experiment</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>Security_Mechanism_Instance</With>
    <Parameters/>
  </FitnessFunction>
  <Experiment>
    <PopulationSize>25</PopulationSize>
    <Generations>100</Generations>
    <MetricWeightManager>
      <Weight name="script_time">1.0</Weight>
    </MetricWeightManager>
    <SelectionProcess>Ranked_Based_Selection</SelectionProcess>
    <Operators>
      <Operator>
        <Name>Single_Point_Mutation</Name> <Weight>25</Weight>
      </Operator>
      <Operator>
        <Name>Two_Point_Crossover</Name> <Weight>75</Weight>
      </Operator>
    </Operators>
  </Experiment>
  <Progams>
    <Progam> <Name>Progam_1</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_2</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_3</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_4</Name> <Parameters/> </Progam>
  </Progams>
</Config>
```

Figure 7.1: Example Experiment for Single Mechanism Process

possibly many times over. However, in most cases the fitness values produced during the experiment run generally consists of too few simulations to be statistically confident in the average/median fitness value for the metrics of an event sequence. To be able to compare event sequences with confidence, an equally large enough sample of

run times for each event sequence is necessary. To do so, the tool is used in its secondary mode, which allows for one event sequence script to be executed a set number of times in a row.

The number of times to simulate an example event sequence should be large enough for confidence in the average script run time produced. A count of ten times is good for a rough idea of the measured results, twenty-five is good for a reasonably confident conclusion, and fifty should be sufficient for statistical confidence to become a significant factor if desired.

It should be noted that if the user wishes to re-create fitness function values, then it is necessary to have the sample metric values created in the experiment. These can either be extracted from the evolutionary search logs, or a new sample can be created within the tool. It will be necessary in this second case to pass in the same parameters that were used in the original experiment to produce similar sample distribution metric values. The second method is recommended, since it increases the consistency of results to create the comparative sample at the same time as the example event sequence evaluations.

Each example event sequence extracted from the exploratory search experiments is simulated against every one of the security mechanism instances. These simulations produce a raw script time for each of the virtual machine instances with and without the security mechanism installed. If desired, a calculated fitness value can be produced either using existing sample values or from sample values created before the simulation.

Figure 7.2 is an example of a configuration file to simulate an example event sequence.

Each different event sequence will produce different raw script times for the virtual machine with and without a security mechanism installed. In general, the comparative

```
<Config>
  <Name>Single_Security_Mechanism_Test</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>Security_Mechanism_Instance</With>
    <Parameters/>
  </FitnessFunction>
  <Test>
    <Script>Event_Sequence_1</Script>
    <Count>25</Count>
  </Test>
  <Progams>
    <Progam> <Name>Progam_1</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_2</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_3</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_4</Name> <Parameters/> </Progam>
  </Progams>
</Config>
```

Figure 7.2: Example Test for Single Mechanism Process

process completed here is interested in the average of these values representing the expected script run time.

The average raw script times for the virtual machines with the security mechanism installed can be examined as a direct value to get a graspable scale of the performance loss. However, it is generally more useful to consider the efficiency loss as a relative measure by comparing the average script time for the secured system against the fresh unsecured installation. For example, the unsecured system may take 10s and the secured system 15s. This can be considered as it took 5s longer, or be considered as taking 150% as long. The second method allows for a relative comparison. For example, 5s of additional time for the secured system is considered differently if the unsecured system took 20s than the case where the unsecured system took 10s.

Considering these raw scores as relative to the unsecured system can be sufficient in some cases. The previous comparison can determine which of the event sequences took a long time to complete, and which took the longest relative to if we had not installed the security mechanism. However, it is not clear if this increase in time was just a general delay created by the load of having the security mechanism installed, or if there is a particular unique additional delay as the result of the characteristics of the individual events in the sequence. Additionally, this method cannot be used if more metrics have been included in the search than just script execution time.

It is generally more accurate and useful to consider each event sequence in regard to its relative fitness function value. The resulting number is not as immediately understandable as the raw times or values seen previously, but can provide additional revealing information. As mentioned before, to get this value it is necessary to use the metric sample distribution values that were created during the experiment that produced the event sequence, or have the re-simulation reproduce a similar sample to perform this fitness function calculation.

The relative fitness function value compares the metrics that make up the fitness function, in this case the script time, relative to a sample distribution created from event sequences of the same length executing with the same security mechanism installed. The reasoning for comparing the event sequences in this manner is the same reason for the choices made in the fitness function construction. Higher fitness function values tell us that the event sequences script time (or other metrics) is at a more extreme location in the distribution of the secured system than in the distribution of the unsecured system.

If all event sequences had the same delay as the result of the installation of the security mechanism, then the distribution of the script times would in effect just be shifted upwards. A fitness function relative comparison would show this delay as

canceling out to a value near zero. However, if a particular event sequence was more negatively affected, then this will become apparent when the fitness function relative comparison shows it as a more extreme z-score change.

For example, consider a distribution of an unsecured system being an average of 10s and a standard deviation of 1s, and the distribution of the secured system being an average of 15s and a standard deviation of 1s. In this case, a time of 15s for the secured system and 10s for the unsecured system is 150% as long. However, compared relative to the distributions, this is a difference of 0 as a z-score. If that secured system time was instead 16s, then it is 160% as long. However, compared relative to the distributions that would be a difference of 1 as a z-score.

The fitness function of course follows this method. If the fitness function had just used the secured system script time, then the search would just find an event sequence filled with actions that take the longest to complete. This sequence might take just as long for the unsecured system. If the fitness function had just used a relative raw script time, then the search would find an event sequence with actions that take longer to complete in the secured system rather than the unsecured. It can be noted that neither of the non-fitness methods can combine multiple metrics. The relative fitness value method allows for the incorporation of multiple metrics at a time through the distribution normalization process. Additionally, the relative comparison indicates which event sequences were delayed due to particular events.

With these different methods of comparison, the user can examine the consequences of choosing each security mechanism. If a raw script time is unacceptable in length, then that mechanism can be removed from consideration. The construction of the event sequence can be examined to see if there is anything unique about the event sequence that should be considered. For example, does one event occur an unrealistic number of times? If so, the search process could be modified to avoid this type of

event sequence.

Next, mechanisms can be compared using the relative script times. Each event sequence produced for one mechanism will have a value with the security mechanism from the search that produced it, and a value for each of the other security mechanisms. If the worst examples found for a search for one mechanism are delayed equally or worse for each of the others, then this mechanism should be noted as performing well. The least delayed of these would be ranked first. If the worst examples for a search for one mechanism are delayed uniquely for only the mechanism whose experiment produced the examples, then this mechanism should be noted as to be avoided. For mechanisms that will likely be avoided, we can rank them on the relative size of the delay.

The final order of choice for a user is to pick one of the well-performing mechanisms unless the features are not broad enough. If there is not one of these, then the user should select one of the remaining mechanisms, starting first with the one whose delayed example event sequences were the least delayed. If the user is interested in more protection than any one security mechanism provides, then they can move on to the next process that considers multiple security mechanisms.

7.3 Search Process for Multiple Security Mechanisms

Goal: From a set of security mechanisms that provide parts of the desired security coverage, select a subset of security mechanisms that provides the most robust security-in-depth coverage while limiting the performance cost. Perform this selection by judging performance cost based on how the installation of the mechanisms negatively affects the performance of the system. This performance will be measured in regard to the user's usage profile, which defines the programs and interactions the user expects to use regularly.

The initial steps for this process are similar to that of the single security mechanism process. The previous process allows for the set of security mechanism under consideration to be reduced to a much smaller group. Security mechanisms from this smaller group of well-performing mechanisms can be added into growing subsets and their results considered to see if this larger set still provides acceptable performance.

Steps: Select a Subset of Security Mechanisms

1. Collect the installation software for each of the security mechanisms.

2. Collect the installation software, and accompanying files, for each of the programs that form the user's usage profile.

3. Set up the experimental environment for the exploratory search tool. Create the virtual machine instances for the fresh installation state without security mechanisms. From this fresh installation, create one virtual machine instance for each security mechanisms being compared. More detailed virtual machine setup and code configuration was described in the previous chapter.

4. Create and run an evolutionary search for event sequence experiment for each of the individual security mechanisms.

5. Explore the search results to select representative event sequences produced by each experiment.

6. Validate the event sequences by evaluating them multiple times with the security mechanism that created them.

7. Evaluate the event sequences multiple times for each of the other security mechanisms.

8. Compare and contrast the quantitative differences in event sequence performance between the security mechanisms to get a ranking of mechanisms. This ranking

will follow that of the process in the previous section.

9. Create sets of multiple security mechanisms that provide the desired security coverage by selecting from the best performing in the required coverage areas.

10. Repeat Steps 3-7 from this process for these subsets instead of for singular mechanisms.

11. Compare and contrast the quantitative differences in event sequence performance between these subsets.

12. Create greater subsets for security-in-depth from existing subsets by adding in additional security mechanisms.

13. Repeat previous 3 steps until satisfied with the security coverage and accompanying performance cost of the final subset of security mechanisms.

7.3.1 Creating Experiments

There will be one experiment created each time a subset of security mechanisms needs to be evaluated. These experiments will be much the same as those created in the previous process for single security mechanisms. Each of these experiments will be identical except for the virtual machine instance $S^?$ used to represent the system state with the security mechanism subset installed.

Previously, only the time it took for an event sequence to complete was of interest. The challenge of security-in-depth is more likely to be a concern of a system administrator or expert who is aware of the other metrics that can be examined about the system's operation. Therefore, the quantification of event sequence performance will be expanded to include these multiple metrics. However, if desired, this process could be completed as was done in the previous section by just considering script time.

The broad experiment parameters remain the same as the previous process. The programs chosen for the experiment are based on those that represent the user's usage profile. The main change is that instead of just having a single metric in the weight manager, there will now be entries for the additional metrics being considered. For example, the most likely categories are additional metrics related to CPU usage, network usage, drive usage, and memory usage. The weights will be chosen to indicate which categories should drive the search direction. One method is to give each category equal weighting of 1.0, and each metric within the category an equal portion of this weight.

Figure 7.3 is an example of the configuration file for one of the experiments. Like before, once this experiment completes, the log files associated with the experiment will contain the top 100 event sequence examples for different categories of behaviour. The event sequences that occur in the most often incomplete or crashed categories are examined using the process in Section 7.4 of this chapter.

7.3.2 Quantifying Results

Like the previous process, the user again can make use of the secondary mode of the tool to produce quantitative values to be used in decision making. The number of times to execute each event sequence example should be enough for confidence in the results produced.

As in the previous process, each example event sequence extracted from the experiments is simulated. During the comparison of single security mechanisms, this process is the same as in the previous process. That is, each security mechanism is compared against all the other single security mechanisms. However, rather than selecting just the best security mechanism, the ranking of the mechanisms is considered. Initial minimal subsets of the best performing mechanisms that fulfill the desired Security-in-Depth coverage are then assembled in subsets to be compared.

```
<Config>
  <Name>Single_Security_Mechanism_Experiment</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>Security_Mechanism_Instance</With>
    <Parameters/>
  </FitnessFunction>
  <Experiment>
    <PopulationSize>25</PopulationSize>
    <Generations>100</Generations>
    <MetricWeightManager>
      <Weight name="script_time">1.0</Weight>
      <Weight name="metric_two">0.5</Weight>
                      <Weight name="metric_three">0.5</Weight>
    </MetricWeightManager>
    <SelectionProcess>Ranked_Based_Selection</SelectionProcess>
    <Operators>
      <Operator>
        <Name>Single_Point_Mutation</Name> <Weight>25</Weight>
      </Operator>
      <Operator>
        <Name>Two_Point_Crossover</Name> <Weight>75</Weight>
      </Operator>
    </Operators>
  </Experiment>
  <Progams>
    <Progam> <Name>Progam_1</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_2</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_3</Name> <Parameters/> </Progam>
    <Progam> <Name>Progam_4</Name> <Parameters/> </Progam>
  </Progams>
</Config>
```

Figure 7.3: Example Experiment with Multiple Metrics

When comparing subsets of security mechanisms, each new subset is compared against the other new ones and against the best performing previous subset. If the relative fitness values for a subset are unacceptable, then it is removed from future

consideration. Otherwise, larger subsets with greater security-in-depth are compared as security mechanisms are added to the best performing subset. This process halts once the desired depth of security coverage is reached, adding additional mechanisms produces unacceptable performance consequences, or there are no longer any more security mechanisms to add.

7.4 Verifying Extreme Emergent Misbehaviour

During the execution of the previous exploratory search experiments, there are sometimes occurrences in which the execution of an event sequence script does not complete, or the virtual machine management process has some form of crash. A crash is generally associated with being unable to start the virtual machine, run a script, close the virtual machine, or reset the virtual machine. It is possible that any of these circumstances are the result of behaviour that is of interest to the person performing the search. On the other hand, it is also true that host system scheduling conflicts, or other management issues, may have caused these aberrational results.

The occurrence of these situations, and the event sequence associated with them, are tracked by the search process. The search process uses a genetic operator to periodically introduce the most commonly occurring of these into the next genetic algorithm population. This maintains interesting event sequences so that descendants can be made from them, and so they can be re-evaluated and the repeatability of their behaviour examined. However, these instances are often not quantified in the regular fitness function, since no fitness value can be created without the performance metrics created after a successful script completion. This removal is done using the weights defined for the fitness function. These incomplete or crashed event sequences are tracked in separate lists ranked by the rate of the incomplete or crashing behaviour.

After the simulation is complete, these event sequences must be examined. The

re-evaluation steps used in the previous processes can be repeated. But in this case, rather than needing metric results for performance quantification, of interest is the ratio of how often the event sequence in question was incomplete or the evaluation process halted due to a crash. If this ratio indicates that the problem is common, then this is a sign that the security mechanism causing this behaviour is not acceptable for use. If the event sequence behaviour in question cannot be reproduced, then it is most likely that the original behaviour was due to chance issues of host system behaviour. Once enough evidence has been gathered through automation, then time can be invested in manually exploring the event sequence's execution.

If an event sequence is determined to be of interest, then it can be further explored by examining sub-sequences with events removed in an attempt to produce a minimal failing example. This is a repetitive process of reviewing sub-sequences and tracking when the removal of an event significantly impacts the ratio of failure. This minimal working example is the most useful example to communicate to the security system creator to examine. Once enough confidence in an example is developed, then it can then be transferred to the similarly configured physical system for confirmation. This last step is the most time-consuming, since it may require the repeated installation and configuration of the physical system.

For the first process described in this chapter, that for a single security mechanism, it is less likely that there will be examples that can be considered as positive examples of extreme emergent misbehaviour. For the second process, which evaluates multiple security mechanisms, these event sequences are expected to be more common. Security mechanisms are rarely designed to work together, and it can be expected that their competition over control points and operating resources will produce examples of emergent misbehaviour.

The described search process can also be used by the creator of a security mecha-

nism. Cases of crashing or incomplete event sequences for a single security mechanism are examples where the creator may have evidence of a design problem. Such cases imply that a security mechanism has a conflict with one of the user's usage profile programs, or the operating system itself. For the case of multiple security mechanisms, the creator is able to examine their security mechanism for emergent misbehaviour while collaborating with other security mechanisms.

Chapter 8

Experimental Results

This chapter explores empirical evaluations that demonstrate the applicability of the method, tool, and processes introduced in the previous three chapters.

In particular, Section 8.1 starts by examining the characteristics of the exploratory search process. These characteristics are explored for search experiments completed using only the metric of script time, and a broader collection of virtual machine system metrics. These experimental evaluations show the ability of the developed fitness function to direct the search to make progress in finding the type of example event sequences the methodology was designed to reveal.

Section 8.2 explores experiments attempting to recommend a single security mechanism for two different usage profiles. The process described in Section 7.2 is followed to compare and contrast example event sequences produced by the evolutionary exploratory search experiments for different security mechanisms.

This is followed, in Section 8.3, by an experiment for comparing subsets of multiple security mechanisms. This is a demonstration of the key steps for comparing subsets of mechanisms found in the process described in Section 7.3. These subsets of security mechanisms consist of enough overlapping security mechanisms such that it is expected that there will be resource and monitoring conflicts.

Both the single and multiple security mechanism experiments will produce example event sequences that fail to complete or, have errors with virtual machine startup and shutdown. The results of these will be briefly discussed in the experiment section that produced them. Section 8.4 explores the process of investigating such examples of emergent misbehaviour following the process from Section 7.4.

To provide a more succinct presentation, this chapter will make generous use of Appendix A to store additional tables and figures related to the experimental results.

8.1 Evaluating the Exploratory Search Process

This section of the experimental evaluation explores the following hypothesis. This hypothesis involves demonstrating that the challenges of system measurement, reproducibility, and variability associated with the design goal of performing an effective evolutionary search are addressed by the developed method. The experimental design to determine if there is support for this hypothesis is given in Subsection 8.1.1, while the results are reported in Subsection 8.1.2, and a discussion of the results completed in Subsection 8.1.3.

Hypothesis 8.1.1. Search Process Progress
The proposed method applying evolutionary search for event sequences to the problem of Effective Security-in-Depth, as described in Chapter 5 and implemented in Chapter 6, results in successful exploratory search experiments. That is, the developed fitness function construction, which integrates multiple metrics, and the search process design, which limits the influence of outliers, allows the search algorithm to find progressively better event sequence solutions in the search space of an Effective Security-in-Depth problem.

To support this hypothesis, this section will explore the sample metric distributions and the metrics/fitness of the population of individuals during each generation of two evolutionary search experiments. Each experiment will consist of the installation of one security mechanism. A variety of metrics will be chosen to guide the exploratory search experiments. First the set of metrics will consist of just script execution time. Then the set of metrics will be expanded to include metrics for hard drive, memory, CPU, and network usage.

8.1.1 Experiment Design

To explore an answer to Hypothesis 8.1.1, two simple single security mechanism experiments, as described in Chapter 7, will be completed. The purpose of each of these experiments is to complete a regular search process. During each completed search process, a wide range of search process data will be produced. This data will include initial sample metric values that define the metric distributions. This data will also include the metric values produced for every solution in each generation of the search process.

The goal of the examination of these metrics is to determine if the design of the search process, in particular the fitness function, allows the search process to make progress towards finding better event sequence examples in the search space. The top example event sequences produced by the search will only be of secondary interest, and examined in terms of their fitness. An exploration of the quality and utility of these examples will be performed in subsequent experiments in this chapter.

A detailed XML configuration file for this experiment can be found in the Appendix as Figure A.1. A full suite of programs will be used to create the usage profile for the experiments. The security mechanism protected instance (AKA the **With** instance) will consist of an installation of **AdAware** Free Anti-Virus 11.12.945.9202.

For the first experiment, only the script execution time metric is used. The second experiment uses a wide-range of metrics as shown in Figure 8.1.

Scripts that complete are evaluated by the fitness function under these five categories of metrics. The first is again script execution time, the last four come from virtual machine statistics/metrics. From the virtual machine, the first category is four hard drive measures: DMA (Direct Memory Access), PIO (Programmed Input/Output), bytes read, and bytes written. The second is four network measures: bytes received, bytes transmitted, receive rate, and transmission rate. The third is

```
<MetricWeightManager>
  <!--Time-->
  <Weight name="script_time">1</Weight>
  <!--HDD-->
  <Weight name="hd_read_bytes">1/4</Weight>
  <Weight name="hd_written_bytes">1/4</Weight>
  <Weight name="hd_dma">1/4</Weight>
  <Weight name="hd_pio">1/4</Weight>
  <!--Network-->
  <Weight name="net_receive_bytes">1/4</Weight>
  <Weight name="net_transmit_bytes">1/4</Weight>
  <Weight name="net_rx">-1/4</Weight>
  <Weight name="net_tx">-1/4</Weight>
  <!--RAM-->
  <Weight name="ram_allocated_pages">1/3</Weight>
  <Weight name="ram_cache_avg">-1/6</Weight>
  <Weight name="ram_cache_min">-1/6</Weight>
  <Weight name="ram_free_avg">-1/6</Weight>
  <Weight name="ram_free_min">-1/6</Weight>
  <!--CPU-->
  <Weight name="cpu_halted">-1/3</Weight>
  <Weight name="cpu_kernel_avg">1/6</Weight>
  <Weight name="cpu_kernel_max">1/6</Weight>
  <Weight name="cpu_user_avg">1/6</Weight>
  <Weight name="cpu_user_max">1/6</Weight>
</MetricWeightManager>
```

Figure 8.1: Experiment Metrics for Search Process Progress

five RAM related measures: allocated pages, RAM cache avg/min, and RAM free avg/min. The fourth is five processor related measures: kernel load avg/max, user load avg/max, and percent of time the CPU spends halted.

Each of the five categories is given a weight of 1.0, and this amount is divided equally among members in each of the categories. Regular weights treat larger values as less desirable to the user, and therefore less desirable to the goal of the search process. Negative weights allow for the smaller values to be treated as less desirable by inverting the fitness function weighting. (Regular XML parsing would require these values to be reported as floating point values, but for ease of reading they have been reported as fractions within their category weightings.)

For hardware emulation the baseline unsecured virtual machine S^0 (AKA the **Without** instance) is a fresh install of Windows XP Service Pack 3 on a 20 GB virtual disk with 2 GB of memory. A minimal setup is completed to enable the execution of event sequences. AutoIT 3.3.12.0 is installed to enable automation scripts to be executed to interact with programs on the system. A physical machine reproduction of this virtual machine instance is used in the final experimental section to verify event sequence examples. This physical machine is a Intel Core 2 Duo, 2.33 GHz with 2 GB of memory and a 20 GB hard drive.

The host machine for all experiments has an Intel Core i7-6850L 3.60 GHz with 64 GB of memory, running Windows 10 Pro with a solid state hard drive. This machine was networked to the Internet with Gigabit Ethernet. Throughout this evaluation chapter, experiments were run with and without parallelization. Without parallelization, only a single virtual machine instance was active at a time. With parallelization, up to four virtual machine instances were active at a time. Each virtual machine is locally hosted on this desktop computer.

As shown in Figure 8.2, the interaction programs installed are: Microsoft Excel/Word 2010 to allow document editing actions, VideoLAN VLC player to allow playing of a 480p video file, and the Java Development Kit 7 to allowing compiling of 117 source code files. The scripts are also able to run DES/3DES/AES256/RC4 encryption actions, FTP remote get and put actions on 2 MB files, file copying actions on a 250 MB directory of files, zip/unzip actions on a 50 MB directory of files, SQL Lite for execution of data base select queries, and web page access through Internet Explorer interactions.

Other variants of these interactions and classes of interactions exist, or are possible, but were not utilized in this experiment. Each program's unique interactions required the installation of the program and the development of the AutoIT script

```
<Progams>
  <Progam> <Name>Word</Name> <Parameters/> </Progam>
  <Progam> <Name>Excel</Name> <Parameters/> </Progam>
  <Progam> <Name>VLC</Name> <Parameters/> </Progam>
  <Progam> <Name>JDK7</Name> <Parameters/> </Progam>
  <Progam> <Name>Crypt</Name> <Parameters/> </Progam>
  <Progam> <Name>FTP</Name> <Parameters/> </Progam>
  <Progam> <Name>FileCopy</Name> <Parameters/> </Progam>
  <Progam> <Name>Zip</Name> <Parameters/> </Progam>
  <Progam> <Name>SQLLite</Name> <Parameters/> </Progam>
  <Progam> <Name>IE</Name> <Parameters/> </Progam>
</Progams>
```

Figure 8.2: Experiment Programs for Search Process Progress

commands necessary to execute them.

There are additional configuration parameters for each program that are omitted from this example configuration. These parameters communicate the locations of the files necessary for certain interactions, and other needed configuration information for individual program actions.

Only a single execution of an experiment with only the metric time and a single execution of an experiment with all the listed metrics are completed. Further experimental runs will be completed in the following experimental sections.

8.1.2 Experiment Results

As previously described, this subsection subdivides the overall experiment into an experiment with just script time and an experiment with the four additional metric categories.

8.1.2.1 Script Time Experiment

When an experiment begins execution, it performs a sample evaluation of individuals of the same event sequence length as the experimental configuration settings. In this experiment, this event sequence length was ten program interaction events. In this

experiment, the sample size was fifty. From the sample, for each metric, a distribution is formed. This distribution is used in the fitness function evaluation to convert an individual's metrics into a fitness value. After this sample distribution was evaluated, the experiment completed a search consisting of a population size of twenty-five for fifty generations. The length of this search, using a single virtual machine individual evaluation thread, was 1.7 days.

This first experiment only uses the metric of script execution time. Figure 8.3 shows a kernel density estimate plot, which is a smoothed version of a histogram. The plot contains script execution time distributions produced by this experiment for the unsecured **Without** virtual machine instance and the secured **With** virtual machine instance. The mean of each distribution is shown by a vertical line. The longer average run time for event sequences once the security mechanism is installed is evident.

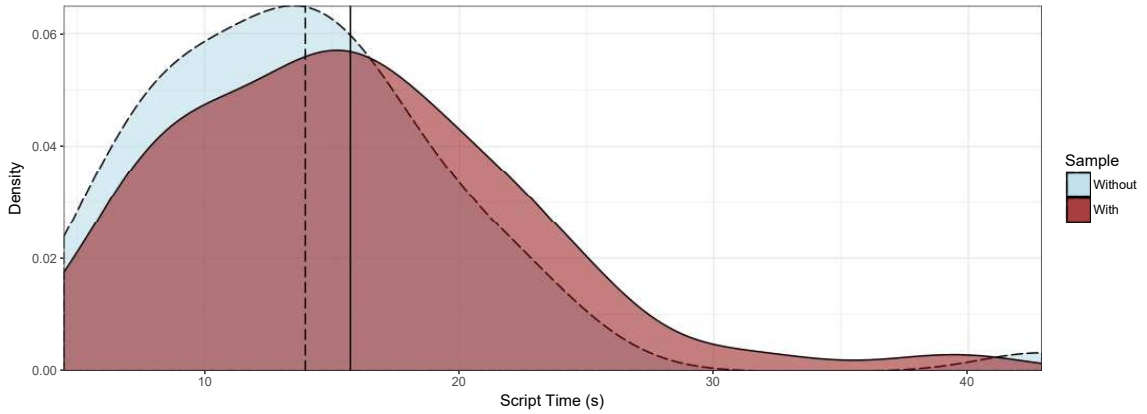


Figure 8.3: Script Time Experiment: Sample Distribution Density

Every experiment begins with a population of random individual event sequences as the first generation. As an experiment progresses, generation to generation, new individuals are created using genetic operators. If an event sequence has been generated before, then the original event sequence is retrieved from the search process history and updated. This update involves the newly created evaluation statistics

being added to the previous existing ones.

Each unique event sequence is given a unique individual ID generated beginning at one and incrementing upwards. Figure 8.4 shows the ID numbers in the population for every generation of the search experiment. Certain ID numbers can be seen repeating across generations. These ID numbers are those of the most fit individuals that are maintained every generation through targeted operators that keep them in the population.

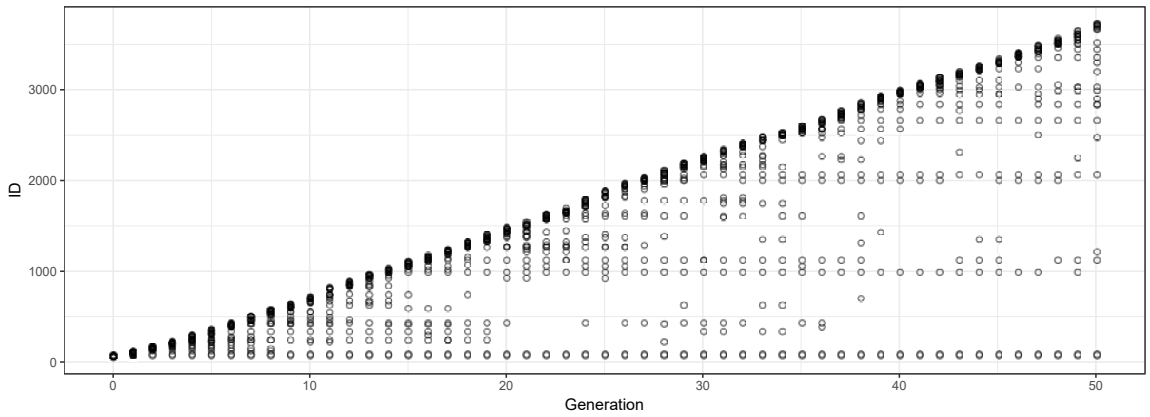


Figure 8.4: Script Time Experiment: Individual ID

In this experiment, there were two unique targeted operators. These operators maintained individuals that had the longest virtual machine startup to shutdown time difference for an event sequence script simulation. One operator propagates the individual with the largest maximum time and another the largest median time. The two most prolific examples of this type of individual in this experiment occurred at ID 76 and 87. These individuals can be seen repeating across the bottom of the ID plot. These individuals were maintained in the population, but their fitness remained at a more average median fitness values of 0.405 and 0.261. It appears some host system event occurred during an early evaluation that disrupted the start/shutdown of the virtual machine, resulting in an outlier result. However, this disruption behaviour did not meaningfully impact the actual script execution time of the associated event

sequences.

As the experiment continued, each individual in each generation was evaluated producing a script time metric for both the secured and unsecured system. In order to direct the experiment's search process, these values are turned into a fitness value. The experiment uses the distribution from Figure 8.3 to judge the script time for each configuration relative to its distribution. The resultant z-scores are then compared. If the z-score (script time location relative to distribution) of the secured system is more positive (slower) than that of the unsecured system, then the result is a positive fitness value. When there is a single metric with a weight of 1.0, such as in this experiment, the fitness is effectively a relative z-score comparison between secured and unsecured system configurations.

Figure 8.5 and Figure 8.6 show the median and maximum fitness across every evaluation of an individual, for each individual in every generation. The line in these figures is created via standard linear regression. It can be noted that the largest median fitness in a generation spikes each time a new maximum fitness is evaluated. However, the largest median fitness in each generation is variable. The median fitness will start high if the first evaluation of an individual is an outlier, but will regress downwards over following generations if this value is not repeatable. This median fitness is used for the rank-based selection process used by the genetic operators.

The two event sequences (ID 76 and 87) can be seen in Figure 8.5 at their median fitness values. The search moved past any value they may have had for the fitness function rather quickly. The two unique targeted operators designed to measure script time from startup to shutdown of the virtual machine don't appear to have value for this search and could be disabled in future experiments. They were left in this search to demonstrate why execution script time was chosen over externally measured startup to shutdown time. On the other hand, it is possible, for other search

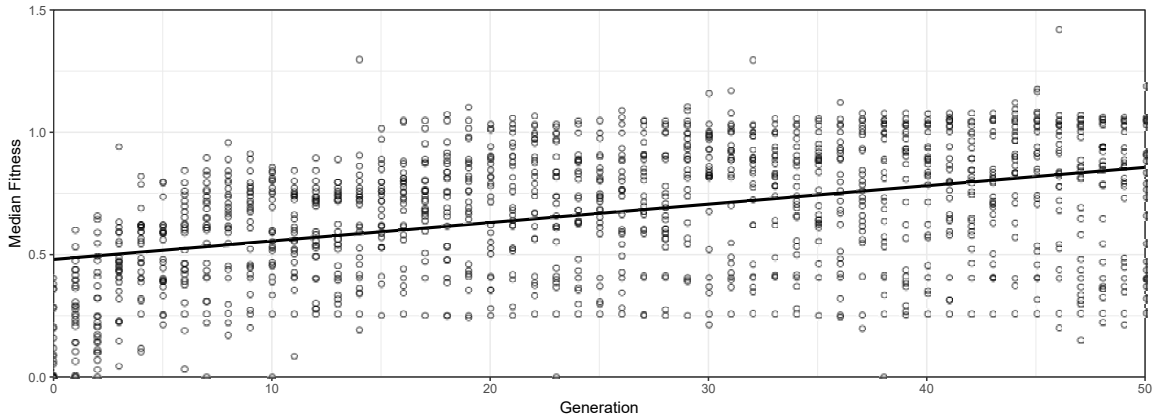


Figure 8.5: Script Time Experiment: Individual Median Fitness

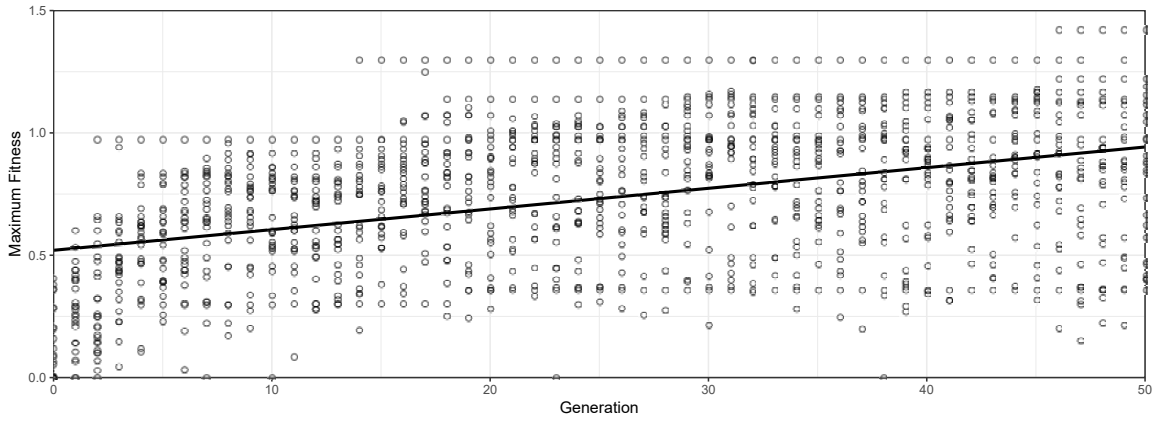


Figure 8.6: Script Time Experiment: Individual Maximum Fitness

goals, that security mechanism disruptions that disturb the startup and shutdown of the virtual machine instances could be of interest.

Between the two figures it is possible to see when a new event sequence is discovered with a large initial max fitness value. This can be seen in sudden spikes in the local maxima of the largest median/max fitness for an individual in the population of a generation. However, although the largest fitness individual is propagated to a following generation, subsequent evaluations are often less extreme and the individual's median fitness drops to a more confident value. This can be seen in the difference between the median/max fitness plots where the median fitness drops, but the maximum is maintained. A search guided by only an outlier max fitness for an individual

is likely to be distracted from its search goal. This distraction is why the maximum was not chosen to guide the search process

To better visualize the population at each generation, Figure 8.7 and Figure 8.8 show the same information as the previous two figures but as population measures instead of for each individual.

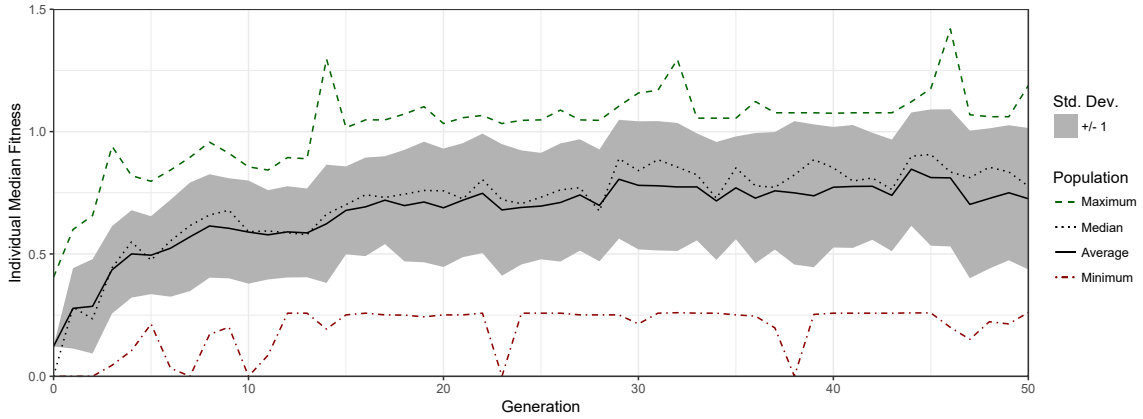


Figure 8.7: Script Time Experiment: Population Median Fitness

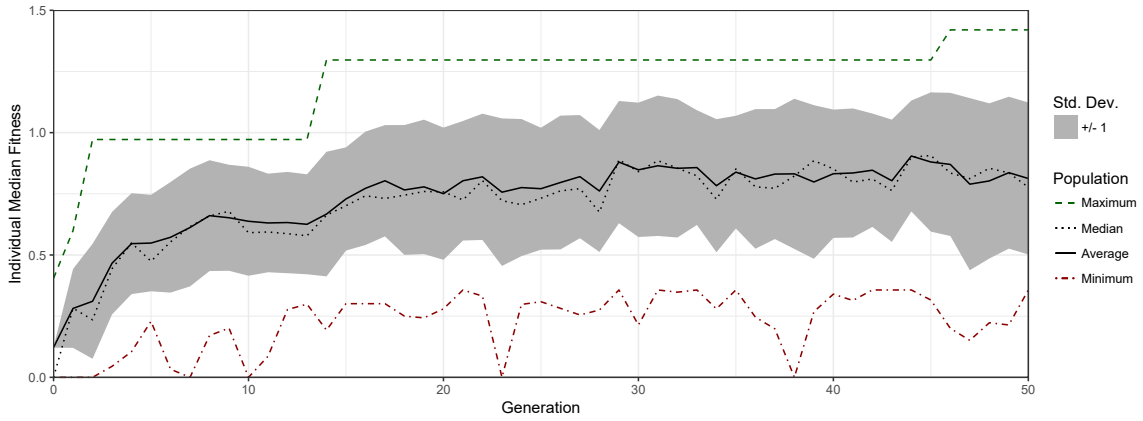


Figure 8.8: Script Time Experiment: Population Maximum Fitness

This experiment was designed to demonstrate the progress of the search process. However, limiting the search to one metric does not fully explore the capabilities of the designed method. In particular, the fitness function design was chosen to normalize and combine a variety of different metrics. The next experiment performs the same search process but with a full suite of metrics being combined in the fitness function.

8.1.2.2 Full Metric Suite Experiment

As seen previously, each metric in the full suite experiment requires a sample distribution to be compiled from an initial sample evaluation of individuals. Again, for this experiment, fifty individual event sequences of the same length as the experiment were evaluated for the sample. For each of the metrics, a distribution was created. These density plots are reported in the Appendix density plots from Figures A.2 to A.6. Similarly, an ID plot as seen in the previous experiment is also in the Appendix. The search was completed for a population size of twenty-five for one hundred generations. The length of this search, using a single virtual machine individual evaluation thread, was 2.84 days.

Not all of the metrics used in this experiment differ notably between the unsecured and secured systems. In particular, at least for this security mechanism, the sent/received bytes for network are an example of a metric where the security systems operations did not change in a meaningful way how many bytes were moved. The metrics where the secured system has a lower distribution mean than the unsecured system, are examples where the metric weighting in the configuration will be negative to invert the relative *diff* comparison for the fitness function.

In each of these metric categories, the metric weighting is applied such that the total weight for a category is 1.0 and each category member takes a proportional sub-weighting. Related metrics within a category that likely duplicate measurement, such as with Cache Minimum and Average, split the weighting further.

Figures 8.9 to 8.13 show the portion of the fitness for each category of the fitness function and for each individual evaluated in each generation. Again each includes a standard linear regression line. RAM and Network fitness values are the least influenced during the search process. The program interactions are in general moving the same amount of data regardless of the installation of the security mechanism.

CPU, HDD, and script time are evidently more influential over the search process.

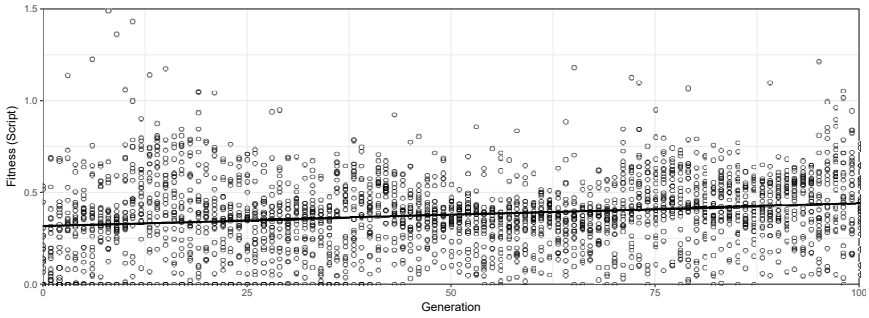


Figure 8.9: Multi-Metric Experiment: Individual Script Time Fitness

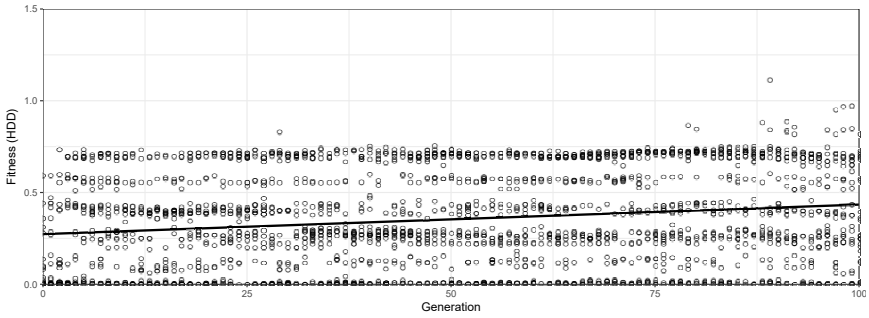


Figure 8.10: Multi-Metric Experiment: Individual HDD Fitness

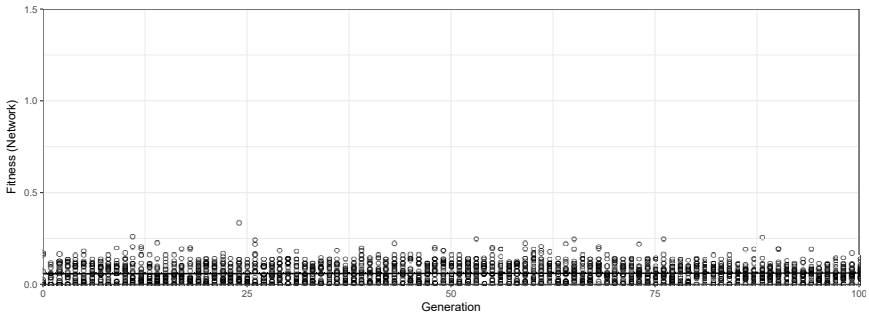


Figure 8.11: Multi-Metric Experiment: Individual Network Fitness

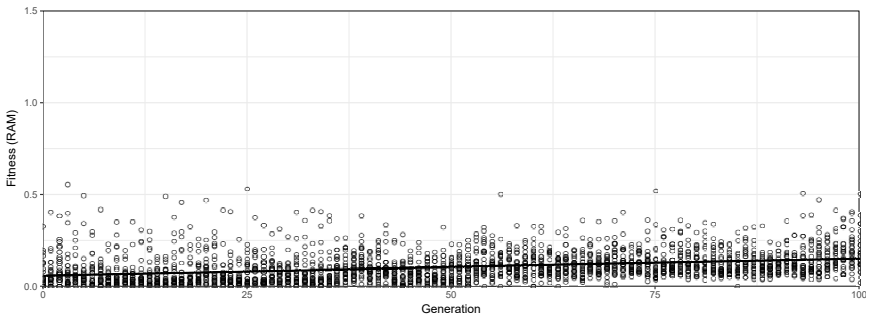


Figure 8.12: Multi-Metric Experiment: Individual RAM Fitness

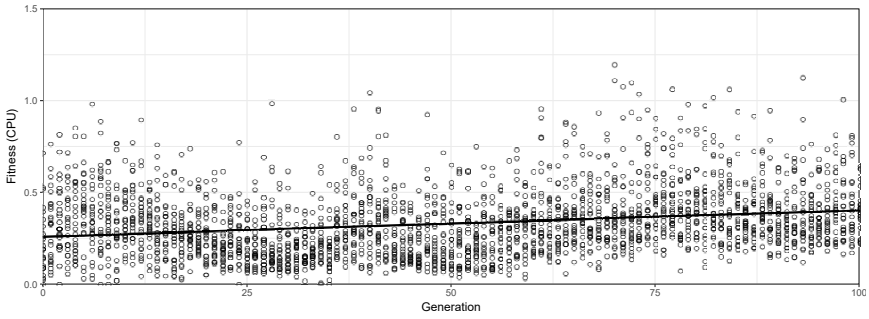


Figure 8.13: Multi-Metric Experiment: Individual CPU Fitness

Figure 8.14 shows each individual's final fitness resulting from the summation of the previous five categories.

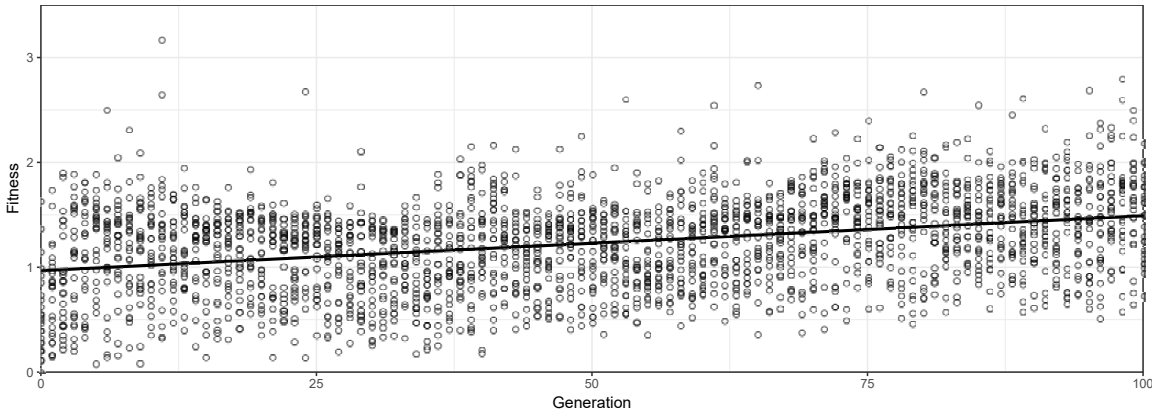


Figure 8.14: Multi-Metric Experiment: Individual Last Evaluation Fitness

Instead of showing only the current evaluation result for a generation, like in the previous figure, Figure 8.15 shows the median fitness of each individual in every generation.

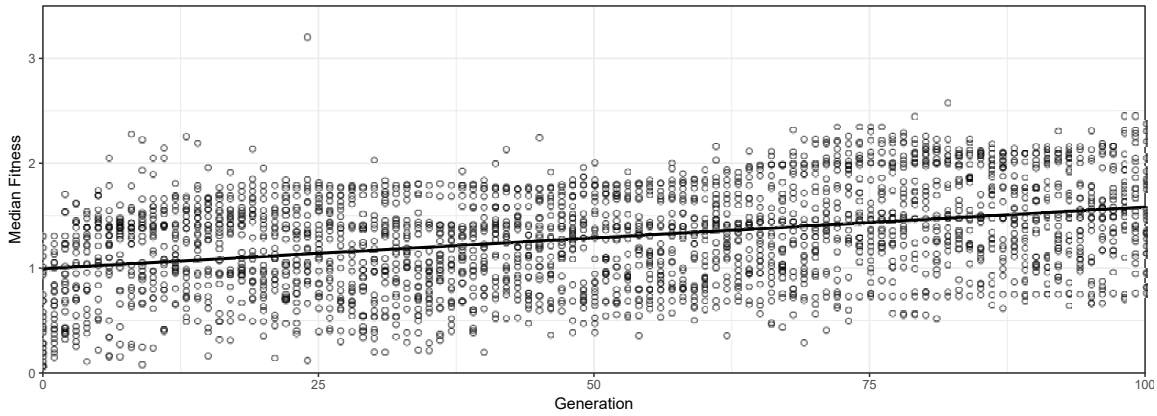


Figure 8.15: Multi-Metric Experiment: Individual Median Fitness

Finally, Figure 8.16 reports the population measures for each generation.

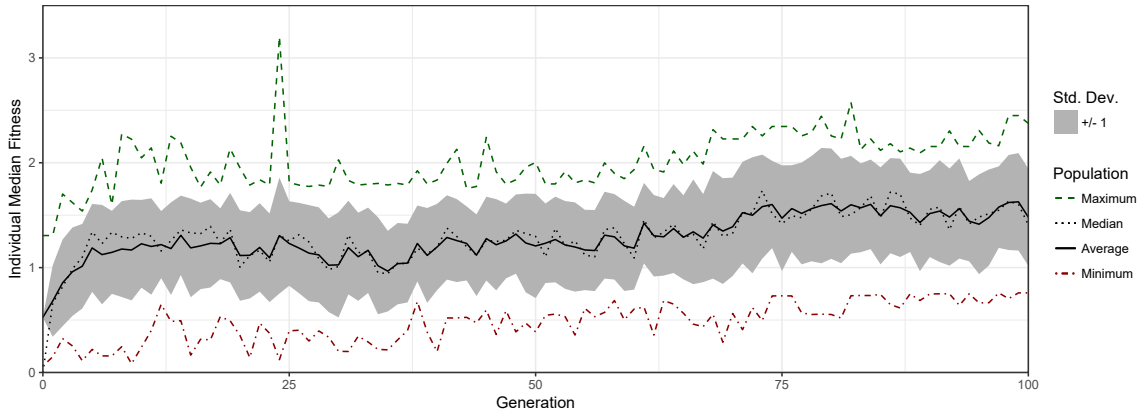


Figure 8.16: Multi-Metric Experiment: Population Median Fitness

8.1.3 Experiment Discussion

The experimental hypothesis, Hypothesis 8.1.1, was that the proposed evolutionary search method described in Chapter 5 and implemented in Chapter 6 would be able to make progress for Effective Security-in-Depth problems. In particular, that it would make progress despite the challenges of different metrics and variable evaluation results such as extreme outliers.

Two experiments were completed. The first experiment was completed for a single metric of script time, for clarity on how the method operates. The second experiment

was completed with a suite of metrics, to show that the fitness function was able to combine them and remain effective.

Both experiments showed that there is danger in the maximum fitness value and that the choice of the median fitness was a preferred solution. Individuals are often evaluated with a maximum fitness that exceeds the highest median fitness in the population. These hard to reproduce maximum fitness values are often the result of scheduling on the host system on which virtual machine is running. This rare increase in fitness should be limited in its influence on the direction of the search process. Instead of using the first or maximum fitness of an individual, the median of its evaluations is used instead. To propagate individuals with possibly interesting extreme fitness values, in case they are repeatable, targeted genetic operators are used instead.

The first experiment visualized the sample distributions for the metric of script execution time. The fitness function calculation uses these distributions to create a fitness value from the script time produced as a result of an individual's evaluation. The following plots explored a search process of fifty generations to demonstrate that the best median fitness increased over the process of the search as more fit individuals were found.

The second experiment completed the same process but for a larger suite of metrics. The result of the fitness function categorizing these metrics and combining them into a final single fitness value was also explored. Plots examined the search process of one hundred generations to demonstrate that the median fitness value increased over the process of the search as more fit individuals were found.

These two experiments support Hypothesis 8.1.1, indicating that the methodology developed in this thesis is suitably designed for making progress finding solutions for the Effective Security-in-Depth problem.

In order to determine the utility of the process, not just its ability to make progress, the produced best median fitness example event sequences must be explored. The experiment in the following section does this by completing a range of single security mechanism experiments for two different usage profiles, and the single metric of script execution time.

8.2 Selecting a Single Security Mechanism

This section follows the same structure for exploring a hypothesis. The new hypothesis for this section involves demonstrating that the process given in Section 7.2 is effective at informing a choice between individual security mechanisms. As before, this experimental section is divided into three subsections for design (Subsection 8.2.1), results (Subsection 8.2.2), and a discussion (Subsection 8.2.3).

Hypothesis 8.2.1. Selecting a Security Mechanism for Usage Profile
The process described in Section 7.2 is effective at informing a choice between single security mechanisms based on a user's usage profile. This search process uses the evolutionary search for event sequences for Effective Security-in-Depth method described in Chapter 5 and implemented in Chapter 6.

To support this hypothesis, the following experimental evaluation explores how the evolutionary search for event sequences method, and the process in Section 7.2, can be used to recommend a security mechanism for a particular usage profile. This recommendation process requires the cross-evaluation of event sequence examples produced by a search experiment for one security mechanism and usage profile against the other prospective security mechanisms.

8.2.1 Experimental Design

To explore an answer to Hypothesis 8.2.1, two different usage profiles will be examined for three different single security mechanisms. In total, six search experiments will be performed. From each search experiment, the top five median fitness example event sequences will be chosen. Each of these examples will be evaluated a total of twenty-five times for each of the three security mechanisms, while maintaining the same usage profile. The results of these evaluations will be compared and contrasted. For each usage profile, a recommendation will be made as to which of the three security mechanisms appears to be the best choice.

The two usage profiles in these experiments are formed by dividing the usage profile program set from Section 8.1 randomly in half into two disjoint subsets. For this hypothesis, the experiment will be limited to the single metric of script execution time.

The purpose of this experiment is to demonstrate that the single security mechanism selection process is effective in allowing the user to choose between security mechanisms. The goal is to show that the user is able to make an informed decision between the three security mechanisms for each usage profile.

A detailed XML configuration file for this experiment can be found in the Appendix as Figure A.9. The **With** security mechanism instance consists of an installation of one of **AdAware** Free Anti-Virus 11.12.945.9202, **Avast** Free Anti-Virus 17.7.2314, or **ClamWin** 0.98.5. **ClamWin** is a passive security mechanism, meaning that after installation it only provides security coverage via scanning procedures when requested (on-demand). The other two are active mechanisms that scan and monitor the system following installation. The baseline unsecured virtual machine S^0 (AKA **Without**) instance is configured the same as in Subsection 8.1.1.

The two disjoint subsets of programs installed to create each of the usage profiles

are shown in Figure 8.17 and in Figure 8.18. These are the same programs as described in the experiments in the previous section, just split into two equal sized randomly chosen subsets of programs. As before, there are additional configuration parameters for each program that are omitted from this example configuration.

```
<Progams>
  <Progam> <Name>Excel</Name> <Parameters/> </Progam>
  <Progam> <Name>JDK7</Name> <Parameters/> </Progam>
  <Progam> <Name>Crypt</Name> <Parameters/> </Progam>
  <Progam> <Name>FileCopy</Name> <Parameters/> </Progam>
  <Progam> <Name>IE</Name> <Parameters/> </Progam>
</Progams>
```

Figure 8.17: Usage Profile 1: Experiment Programs

```
<Progams>
  <Progam> <Name>Word</Name> <Parameters/> </Progam>
  <Progam> <Name>VLC</Name> <Parameters/> </Progam>
  <Progam> <Name>FTP</Name> <Parameters/> </Progam>
  <Progam> <Name>Zip</Name> <Parameters/> </Progam>
  <Progam> <Name>SQLLite</Name> <Parameters/> </Progam>
</Progams>
```

Figure 8.18: Usage Profile 2: Experiment Programs

From each of the six search experiments, the top five event sequences found, as ranked by median fitness (median script execution time), will be selected. This creates two groups of fifteen example event sequences. Each of the event sequences are re-evaluated as described in the process in Chapter 7.2. This evaluation requires a test execution using the developed tool, which simulates the event sequence twenty-five times for each of the three security mechanisms being compared. Each test will use the same programs (the same usage profile) as the experiment that produced the event sequence example. An example of the re-evaluation configuration file is described in the Appendix as Figure A.10.

8.2.2 Experiment Results

As previously described, this subsection subdivides the experiment evaluation for this hypothesis into an experiment for Usage Profile 1 and an experiment for Usage Profile 2. More detailed tables related to the figures shown, as well as extended tables and figures related to each experiment, can be found in Appendix A. Comments on the information reported will be given, but the majority of comparative discussion will be saved for the following experiment discussion subsection.

Each genetic algorithm search process was completed for a population size of twenty-five and one hundred generations. The search time, using four virtual machine individual evaluation threads instead of the singular one used previously, ranged from 1.92 to 2.36 days. Each re-evaluation of an example event sequence for a single virtual machine instance took on the order of approximately half an hour.

This experimental section will make use of box and whisker plots. Each plot will show a box representing the range from the lower quartile to the upper quartile as well as the median point in between. This box spans the inter-quartile range. Any numbers below the lower quartile or above the upper quartile by 1.5 times the inter-quartile distance are designated outliers. The whiskers on the box extend from the min and max values in the distribution, which are not designated as outliers.

8.2.2.1 Usage Profile 1

Usage Profile 1 consists of Excel spreadsheet editing, Java JDK compiling, file encryption operations, file copying operations, and IE browser web page interaction.

Figure 8.19 reports script execution time for 150 random sample event sequences evaluated for each of the virtual machine instances. These sample measures are necessary for the eventual comparison of the produced example event sequences via fitness values. They also provide a general understanding of the system load created by the installation of each security mechanism.

First, the unsecured **Without** virtual machine instance is sampled. This is followed by the creation of a sample distribution for each of the security mechanism installations. As mentioned previously, **ClamWin** is a passive on-demand security mechanism, meaning that after installation it only provides security coverage via scanning procedures when the user requests it. As a result, the reported **ClamWin** values are expected to be roughly equivalent to the **Without** system. Both **AdAware** and **Avast** are active security mechanisms, scanning and monitoring the system continuously, which create apparent delays for event sequences.

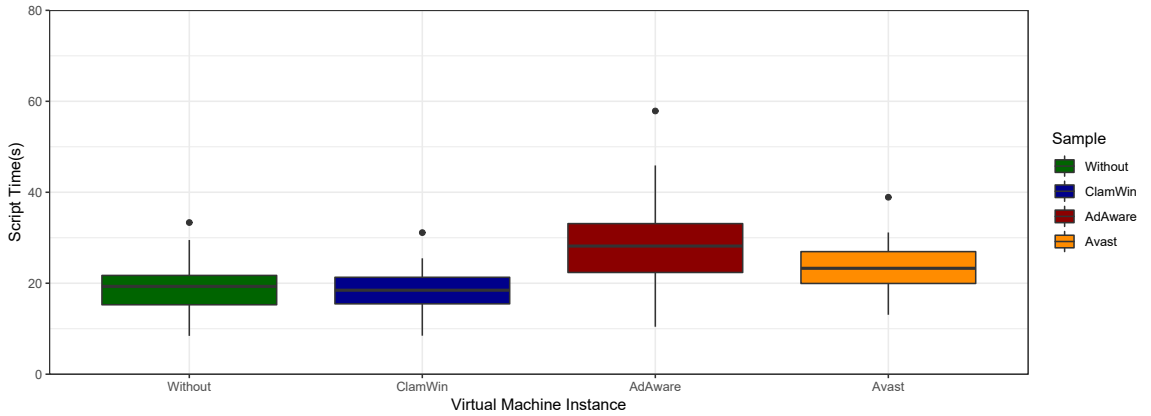


Figure 8.19: Usage Profile 1: Sample m^{time} Statistics in s

A more explicit table of the values depicted in Figure 8.19 can be found in the Appendix as Table A.2. Similarly, in the Appendix, Table A.3 and the box plot in Figure A.11 report the sample script measures reported relative to the **Without** mean time. That is, each script time reported in one of the **ClamWin**, **AdAware**, and **Avast** samples has been divided by the average script time from the **Without** sample before being examined.

The mean time of **ClamWin** can more accurately be seen as relatively similar to that of **Without** at 97.5%. This lower value is within the standard deviation of **Without** system measurement. **AdAware** has a mean time of 152.4% and **Avast** a mean time of 123.1%. Of note, **AdAware** also has almost double the variance of

Avast.

As described previously, one search experiment was completed for each of the three security mechanisms. From each of these experiments, five event sequences with the top median fitness values were selected. These event sequences are reported in Table 8.1. There were two **Avast** incomplete event sequences and these are reported at the end of the table. These two event sequences will be first addressed in the experiment discussion for this section and then examined in the last experimental section of this chapter. The incompletion behaviour noted briefly for these two event sequences during the search process is established as not repeatable under further examination.

Table 8.1: Usage Profile 1: Event Sequence Examples

event_seq	ev_1	ev_2	ev_3	ev_4	ev_5	ev_6	ev_7	ev_8	ev_9	ev_{10}
$clamwin_1$	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0
$clamwin_2$	Copy-0	Copy-0	Comp-1	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0
$clamwin_3$	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Comp-0	Copy-0	Copy-0
$clamwin_4$	Crypt-4	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0
$clamwin_5$	Copy-0	IE-0	Copy-0	Excel-5	Comp-1	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0
$adaware_1$	Copy-0	Excel-0	Copy-0	Comp-1	Copy-0	Copy-0	Empty-0	IE-0	Copy-0	Crypt-5
$adaware_2$	Copy-0	Excel-0	Copy-0	Comp-1	Copy-0	IE-0	Empty-0	IE-0	Copy-0	Crypt-5
$adaware_3$	Crypt-4	Excel-0	IE-0	Crypt-4	Copy-0	Copy-0	Empty-0	IE-0	Copy-0	Crypt-5
$adaware_4$	Copy-0	Comp-0	Copy-0	Comp-1	Copy-0	Copy-0	Empty-0	IE-0	Copy-0	Crypt-5
$adaware_5$	Copy-0	Excel-0	Copy-0	IE-0	Copy-0	Copy-0	Empty-0	IE-0	Copy-0	Crypt-5
$avast_1$	Copy-0	Empty-0	Crypt-5	IE-0	Copy-0	Copy-0	Comp-0	Copy-0	Copy-0	Excel-18
$avast_2$	Excel-1	Copy-0	Copy-0	Copy-0	Copy-0	Copy-0	Comp-0	Copy-0	IE-0	Excel-18
$avast_3$	Excel-1	Copy-0	Crypt-5	Copy-0	Copy-0	Copy-0	Comp-0	Copy-0	IE-0	Excel-18
$avast_4$	Excel-1	Copy-0	Excel-14	Copy-0	Copy-0	Copy-0	Comp-0	Copy-0	IE-0	Empty-0
$avast_5$	Crypt-0	Copy-0	Crypt-5	IE-0	Copy-0	Copy-0	Comp-0	Copy-0	Copy-0	Excel-18
$avast\_inc_{1,1}$	IE-0	Copy-0	Crypt-5	IE-0	Copy-0	Copy-0	Comp-0	Copy-0	Copy-0	Excel-18
$avast\_inc_{1,2}$	IE-0	Copy-0	Crypt-5	IE-0	IE-0	Crypt-7	Comp-0	Copy-0	Copy-0	Excel-18

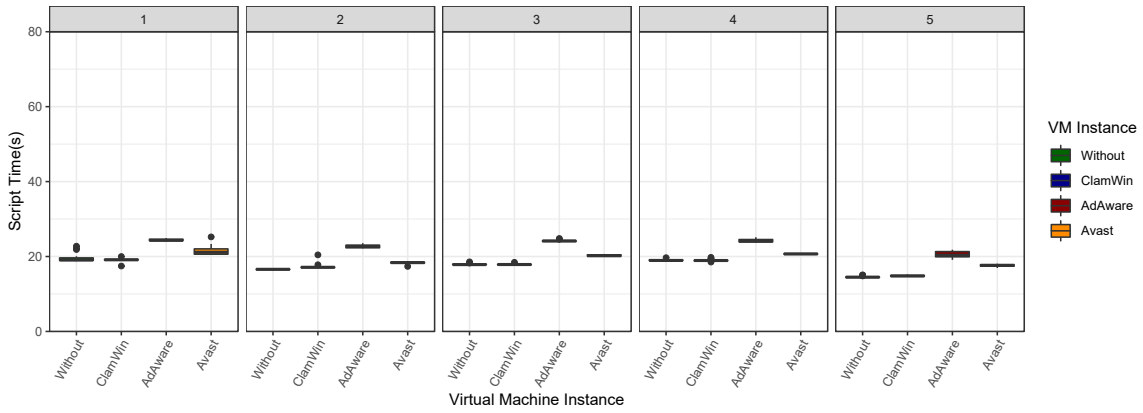
A common characteristics seen across the example event sequences, unique to this usage profile, is the repetition of file copy program interactions dominating the event

sequences. Details for each program-interaction combination can be found in the reference table in the Appendix at Table A.1.

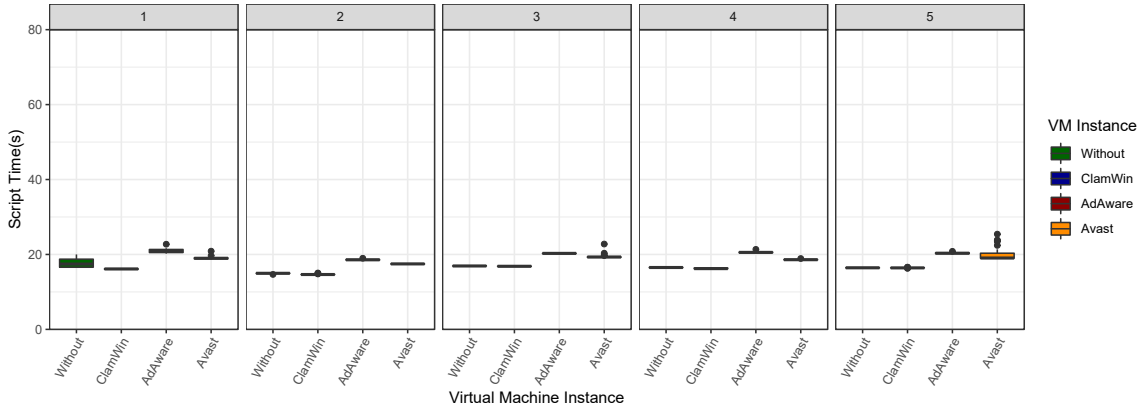
The box plot in Figure 8.20 reports script execution time results from the twenty-five evaluations for each example event sequence. All four configurations of the **Without**, **ClamWin**, **AdAware**, and **Avast** virtual machine instances are reported. This table provides visual evidence of the consistency of the **ClamWin** results being comparable to the **Without** results across the top five event sequences for all three security mechanisms. Even the dedicated test for **ClamWin** did not find any special examples of **ClamWin** being exposed. This is as expected, since **ClamWin** is an on-demand mechanism. At the same time, the consistency of **AdAware** continuing to be slower than **Avast** can be seen, despite two of the three tests targeting a different security mechanism.

In the Appendix, Table A.4 reports the mean and standard deviation numbers for Figure 8.20. The information reported in Table A.4 and Figure 8.20 can be hard to compare. Instead of raw values, in the Appendix, Table A.5 and Figure A.12 show the average time for each event sequence relative to the **Without** time. This conversion emphasizes the information from the previous table.

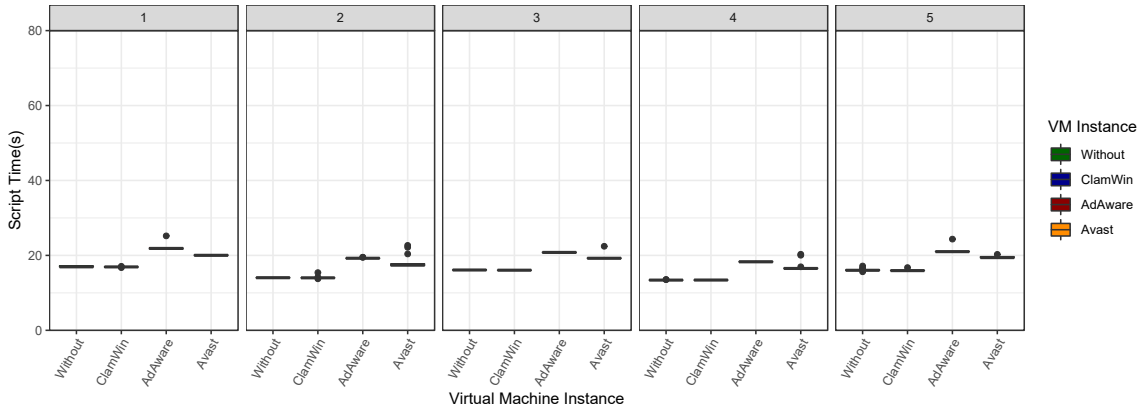
As mentioned in Chapter 7, just comparing relative script times may not actually reveal as much as considering the fitness of event sequences. In part, this was because it is not possible to compare across individual metrics effectively when the search makes use of multiple classes of metrics. Instead, in such searches the fitness value that combines information about all the metrics into one value is used. In these experiments, there is only one metric, so this is not an issue. However, even more importantly, although a script may be affected (in this case made slower) by one security mechanism more than another, this effect may be less extreme relative to its distribution than the effect relative to the **Without** distribution.



(a) ClamWin Experiment



(b) AdAware Experiment



(c) Avast Experiment

Figure 8.20: Usage Profile 1: Event Sequence Mean (St.Dev.) for m^{time} in s

As a result, it is still possible that even the relative information reported in Table A.5 may be deceiving. There is already evidence from the initial samples in Figure 8.19 for Usage Profile 1 that the installation of **Avast** results in performance loss and even more-so with the installation of **AdAware**. As a result, it is known that for some event sequences there will be consistent delays just due to the resource requirements of the security mechanism installation and operation. However, more interesting to us are the event sequences that begin at one z-score (distribution location) in **Without** but are not at an equivalent z-score in the secured virtual machine distribution. In particular, those event sequences that have higher z-scores (worse performance relative to their distribution) versus the **Without** virtual machine instances are of interest. This goal was reflected in the design of the search process fitness function.

The values from Figure 8.19 can be further examined as fitness values. These fitness values in effect reproduce the view the search process had of the event sequence, but with more samples to increase confidence in the values. It should be noted that unlike the fitness values used for the search process, these fitness values are reported without forcing a floor of 0 on negative values. A detailed fitness value comparison relative to the distributions reported in Figure 8.19 can be seen in the Appendix in Table A.6 and Figure A.13. A reduced comparison of this information is shown in Figure 8.21. This comparison only shows one event sequence from each search experiment. This event sequence had the highest average fitness for the security mechanism being tested.

From this fitness comparison, there is a confirmation that the previous judgment that **AdAware** has more negative consequences than **Avast** continues to be supported, even after examining the values as fitness values.

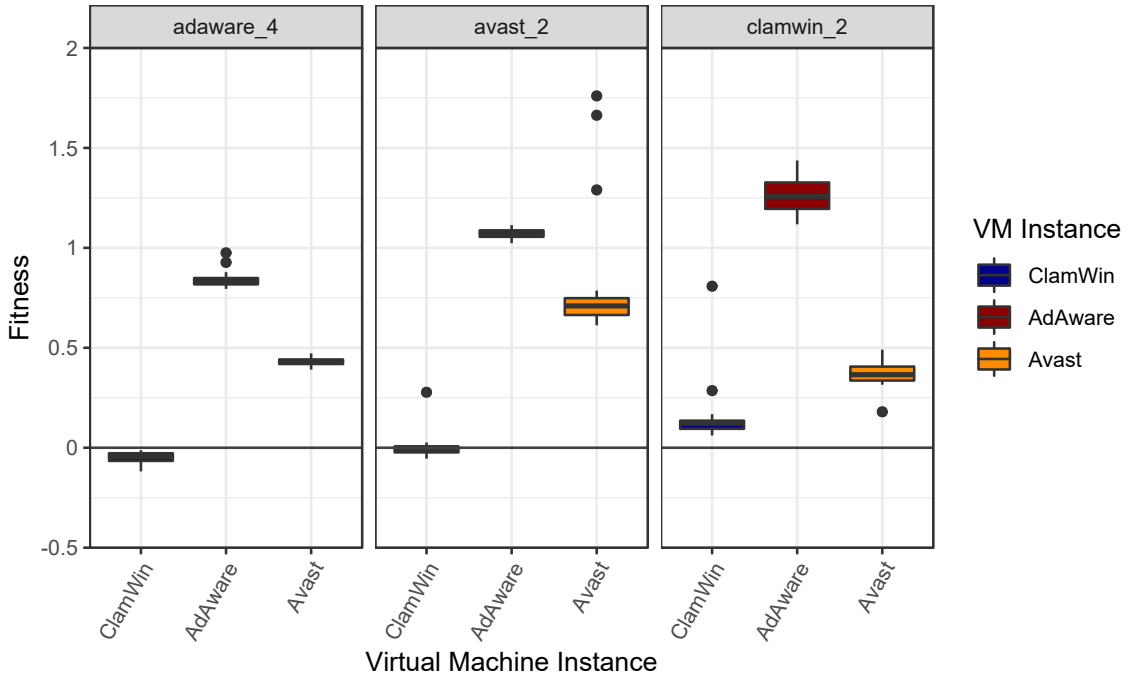


Figure 8.21: Usage Profile 1: Top Event Sequence m^{time} Fitness

8.2.2.2 Usage Profile 2

Usage Profile 2 consists of Word document editing, VLC movie/audio playing, FTP transmission, zip/unzip of compressed files, and SQLLite database interactions.

The structure for this usage profile will follow that of the previous usage profile experiment. As before, Figure 8.22 reports script execution time for 150 random sample event sequences evaluated for each of the virtual machine instances.

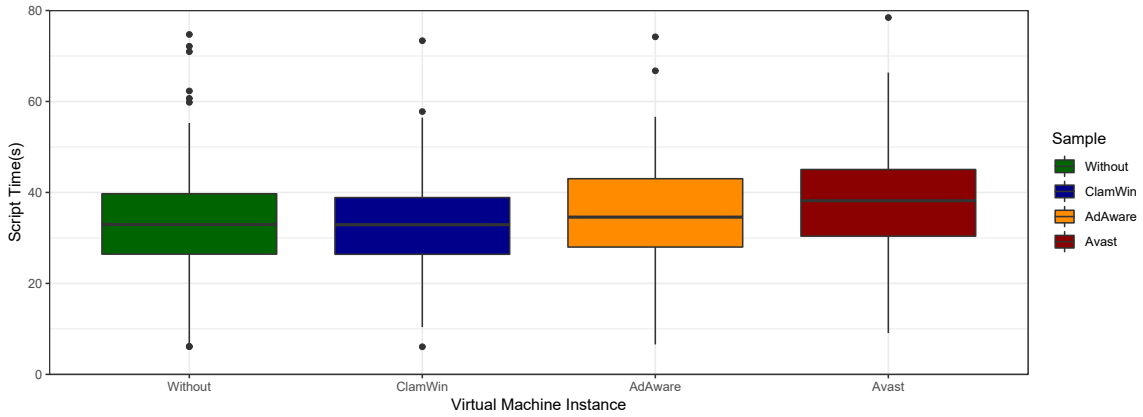


Figure 8.22: Usage Profile 2: Sample m^{time} Statistics in s

Similarly, a more explicit table of the values depicted in Figure 8.22 can be found in Table A.7 in the Appendix. Also in the Appendix, Table A.8 and the box plot in Figure A.14 report the same sample script measures relative to the **Without** mean time. The mean time of **ClamWin** is similar to that of **Without** at 99.7% while **AdAware** is at 108.0% and **Avast** at 116.3%. It can also be noted the variance of **AdAware** is roughly the same as **Avast**.

Table 8.2: Usage Profile 2: Event Sequence Examples

$event\_seq$	ev_1	ev_2	ev_3	ev_4	ev_5	ev_6	ev_7	ev_8	ev_9	ev_{10}
$clamwin_1$	Zip-1	Word-8	Zip-1	Zip-1	Zip-0	Zip-0	Zip-1	FTP-1	VLC-0	Zip-0
$clamwin_2$	Zip-1	Word-8	Zip-1	Zip-1	Zip-0	Zip-1	Zip-1	Zip-1	Zip-1	Zip-1
$clamwin_3$	Zip-1	Word-8	Zip-1	Zip-1	Zip-0	VLC-0	Zip-1	FTP-1	Zip-1	Zip-1
$clamwin_4$	Zip-1	Word-8	Zip-1	Zip-1	Zip-0	Zip-0	Zip-1	FTP-1	Word-1	Zip-1
$clamwin_5$	Zip-1	Word-8	Zip-1	Zip-1	Zip-0	Zip-1	Zip-1	Zip-0	Zip-1	Zip-1
$adaware_1$	SQL-0	VLC-0	VLC-0	Zip-1	VLC-0	SQL-0	Word-8	VLC-0	Zip-1	Zip-1
$adaware_2$	VLC-0	Zip-1	VLC-0	VLC-0	VLC-0	SQL-0	Word-8	Zip-1	Zip-1	Zip-1
$adaware_3$	SQL-0	VLC-0	VLC-0	VLC-0	VLC-0	Zip-1	Word-8	Zip-1	Zip-1	Zip-1
$adaware_4$	VLC-0	SQL-0	VLC-0	Zip-1	VLC-0	SQL-0	Word-8	Zip-1	Zip-1	Zip-1
$adaware_5$	Zip-0	VLC-0	VLC-0	Zip-1	VLC-0	SQL-0	Word-8	VLC-0	Zip-1	Zip-1
$avast_1$	Word-3	VLC-0	Word-16	Word-20	SQL-0	Word-20	Word-8	Word-16	Word-3	Word-13
$avast_2$	Word-3	VLC-0	Word-16	Word-20	SQL-0	Word-20	Word-8	Word-16	Word-3	Zip-1
$avast_3$	Word-3	VLC-0	Word-16	Word-20	VLC-0	Word-20	Word-8	Word-16	Word-3	Zip-0
$avast_4$	Word-3	VLC-0	Word-16	Word-20	VLC-0	Word-20	Word-8	Word-16	VLC-0	VLC-0
$avast_5$	Word-3	VLC-0	Word-16	Word-20	VLC-0	Word-20	Word-8	Word-16	VLC-0	Zip-0
$avast\_inc_{2,1}$	Word-3	VLC-0	Word-16	Word-20	SQL-0	Word-20	Word-8	Word-16	Word-3	Zip-1

As described previously, one search experiment was completed for each of the security mechanisms. From each experiment, five event sequences with the top median fitness values were selected. These event sequences are reported in Table 8.2. There was one **Avast** incomplete event sequence, and it is included at the end of the table. This event sequence $avast\_inc_{2,1}$ is in fact just the event sequence $avast_2$ tracked via a separate listing for incomplete sequences. The incompletion behaviour for this event

sequence during the search process is established as not repeatable under further examination.

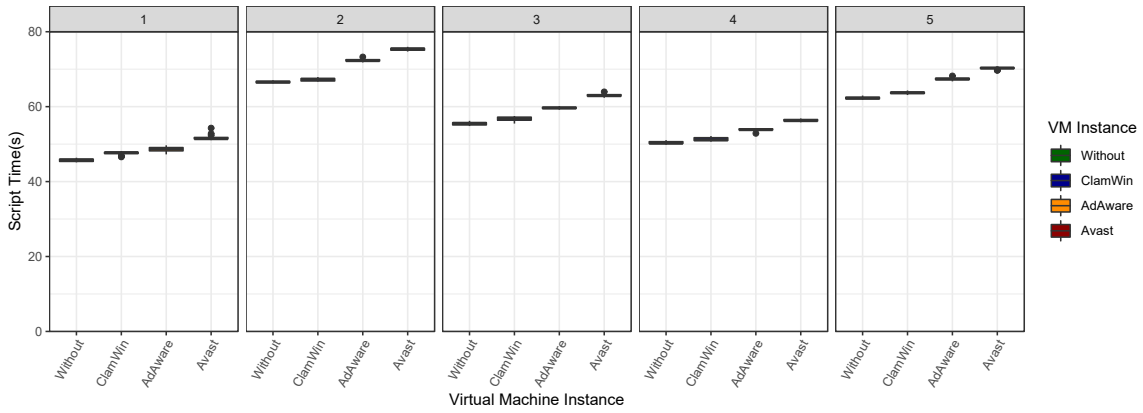
Unique to this usage profile is how each experiment is dominated by a different type of program. **ClamWin** has primarily zip/unzip program interactions. **AdAware** has a mix of VLC, SQL, Word, and Zip program interactions. **Avast** has mostly Word and VLC program interactions. Details of what each program-interaction combination is can be found in Table A.1 in the Appendix.

The box plot in Figure 8.23 reports script execution time results from the twenty-five evaluations for each example event sequence. As expected, the **ClamWin** results are again comparable to the **Without** results. At the same time, it can be seen that **Avast** is slower than **AdAware**, despite two of the three tests targeting a different security mechanism. Evident is the different range of results depending on which security mechanism was being tested. **ClamWin** has long script times, **AdAware** has script times similar to Usage Profile 1, and **Avast** has rather short script times.

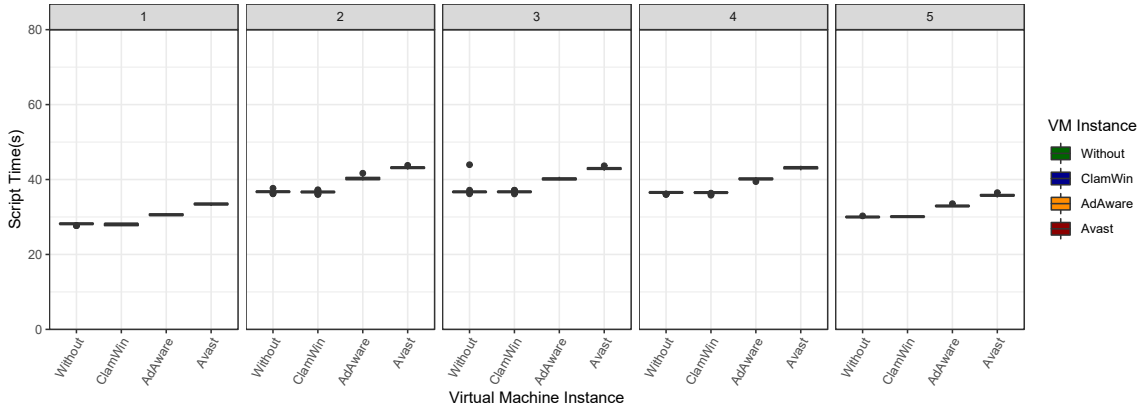
Again, in the Appendix, Table A.9 reports the mean and standard deviation numbers for Figure 8.23 and Table A.5/Figure A.12 report the average time for each event sequence relative to the **Without** time. This conversion emphasizes the information from the previous table.

As discussed earlier in this subsection, it is still possible for the information reported in Figure 8.23 to be deceiving. There is already evidence from the initial samples in Figure 8.22 for Usage Profile 2 that the installation of **Avast** has the largest performance loss, followed by the installation of **AdAware**. However, this difference appears less extreme than for Usage Profile 1 seen previously.

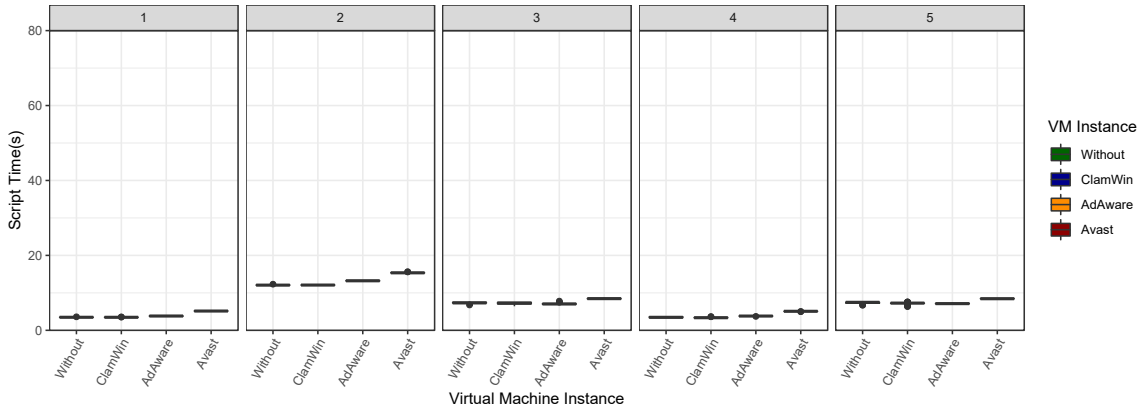
The values from Figure 8.22 can be further examined as fitness values. As before, a detailed fitness comparison relative to the distributions reported in Figure 8.22 can



(a) ClamWin Experiment



(b) AdAware Experiment



(c) Avast Experiment

Figure 8.23: Usage Profile 2: Event Sequence Mean (St.Dev.) for m^{time} in s

be seen in the Appendix in Table A.11 and Figure A.16. A reduced comparison of this information is shown in Figure 8.24. This comparison only shows one event sequence from each search experiment. This event sequence had the highest average fitness for the security mechanism being tested.

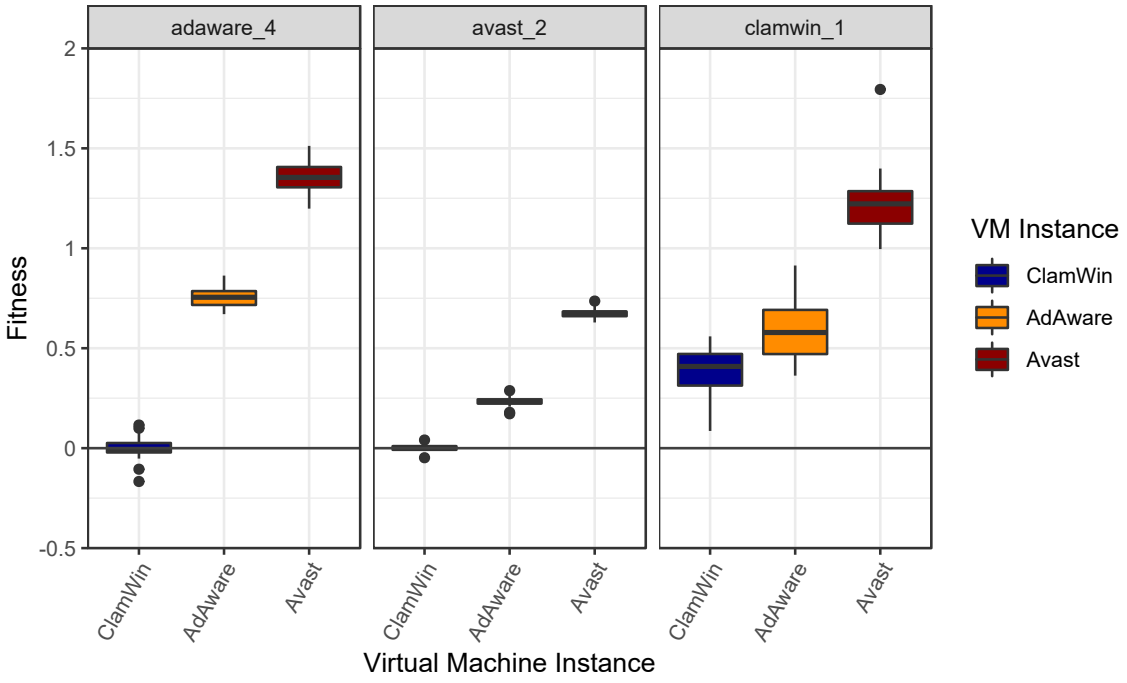


Figure 8.24: Usage Profile 2: Top Event Sequence m^{time} Fitness

From these charts there is confirmation that **Avast** has more negative consequences than **AdAware**. Viewing these values as fitness values also more clearly visualizes the differences between mechanisms. The script times for examples produced by each of the three experiments were noticeably different. However, once converted to a fitness value, the event sequences can be compared on a relative fitness level.

8.2.3 Experiment Discussion

The experimental hypothesis, Hypothesis 8.2.1, was that the results of the developed evolutionary search process could be used to inform the choice between different

security mechanisms depending on the usage profile. Further, this hypothesis was that experimental search processes using the proposed evolutionary search method, described in Chapter 5 and implemented in Chapter 6, could be used to produce challenging example event sequences for each of the security mechanisms. These examples could then be cross-compared using the exploratory search tool and process in Subsection 7.2. This comparison would then allow the user to choose between the mechanisms for each usage profile

Two experiment groups were completed. Each experiment was completed for a unique usage profile consisting of splitting the programs from the experiment in Section 8.1 into two subsets. Then for each usage profile, three experiments were completed for each security mechanism. The top five event sequences were selected from each and twenty-five evaluations completed for each of the three security mechanisms.

Both experiments confirmed the the **ClamWin** anti-virus security mechanism was resource neutral as it requires the user to initiate any security coverage via an anti-virus scan. This was expected, and these results provide a grounding that provides confidence in the results. The experiments also confirmed that both **AdAware** and **Avast** are active anti-virus mechanisms with noticeable resource loads and processing requirements.

Throughout both experiments, the numbers were explicitly described via tables and then graphically explored via box plots. Each usage profile progressed through exploring the numbers as raw script times, a relative comparison to the **Without** virtual machine instance, and finally a comparison as fitness values.

The first experiment began with a sample script time exploration that gave us an initial indication that **AdAware** created a greater slowdown than **Avast**. All three of the event sequence example groups produced for Usage Profile 1 had a similar make-up, with file copying program actions dominating the program interactions. The

explored information progressed through raw values, relative values, and finally fitness values. Raw values were the most accurate description of the results, while relative values were easier to compare, and finally fitness values were the most informative. All three methods of reviewing the results agreed on the conclusion that **Avast** would be the recommended security mechanism for Usage Profile 1 based on performance costs and the desire of active anti-virus coverage.

The second experiment also began with a sample script time exploration that gave us an initial indication that **Avast** created a greater slowdown than **AdAware**, although to a lesser extreme than for Usage Profile 1. All three of the event sequence example groups produced for Usage Profile 2 had unique make-ups, with different programs dominating the program interactions of each. Again, the explored information progressed through raw values, relative values, and finally fitness values. All three methods of reviewing the results agreed on the conclusion that **AdAware** would be the recommended security mechanism for Usage Profile 2.

These two experiments support Hypothesis 8.2.1 that the method, tool, and developed process are suitable for the goal of determining a security mechanism based on a user's individual usage profile. A different one of the two active security mechanisms was, in fact, recommended for each of the two usage profiles.

In all the experiments, there were examples of errors with virtual machine startup and shutdown. However, none were repeatable once the event sequence execution was isolated. These errors are not uncommon when multiple virtual machines are being managed simultaneously during a search process. The majority of errors occur when trying to restore the virtual machine state to a stored past state, before running the current event sequence. Therefore, the failure can be judged as being unassociated with the current event sequences behaviour.

In two of the six experiments, there were examples of incomplete event sequences.

All of these examples occurred in the **Avast** search experiments. There were two examples for Usage Profile 1 ($avast\_inc_{1,1}$ and $avast\_inc_{1,2}$) and one example for Usage Profile 2 ($avast\_inc_{2,1}$). For Usage Profile 1, $avast\_inc_{1,1}$ was incomplete in 1 of 45 executions during the search, and $avast\_inc_{1,2}$ was incomplete in 1 of 101 executions. For Usage Profile 2, $avast\_inc_{2,1}$, which was in fact $avast_2$, was incomplete in 1 of 95 executions. Each of these three, in addition to a number of others not listed, also had virtual machine instance restore errors.

The incompletion behaviour in the three of these did not end up as repeatable when examined in the final experiment in Section 8.3.2. There was initial evidence of this when none of the re-evaluations of $avast_2$ were incomplete for the script execution time comparisons. All re-evaluations were completed as a single thread process with a single virtual machine instance active at a time. The comparison in the final section found no incompletions for twenty-five re-evaluations of each of these three event sequences.

8.3 Examining Subsets of Security Mechanisms

As before, this section has a new hypothesis to explore. This hypothesis involves demonstrating that the process given in Section 7.3 is effective at informing a decision when selecting a subset of security mechanisms. In other words, is the research completed in this thesis effective as a solution to the problem of Effective Security-in-Depth. This section is structured as seen previously with subsections for design (Subsection 8.3.1), results (Subsection 8.3.2), and discussion (Subsection 8.3.3).

Hypothesis 8.3.1. Evaluating Subsets of Security Mechanisms
The process described in Section 7.3 is effective at informing a choice between subsets of security mechanisms based on a user's usage profile. This search process uses the evolutionary search for event sequences for the Effective Security-in-Depth method described in Chapter 5 and implemented in Chapter 6.

Similar to the previous experiment for single security mechanisms, this experiment will perform three evolutionary searches for interaction event sequences experiments. Each of the three search experiments consists of an increasingly larger subset of security mechanisms. The first subset consists of four mechanisms, the next five, and finally six security mechanisms. The fitness function progress, the top median fitness results, and any examples of unwanted emergent behaviour will be explored to inform a comparative decision between the increasingly larger subsets of security mechanisms.

8.3.1 Experiment Design

To explore an answer to Hypothesis 8.3.1, three experiments as described in Chapter 7.3 will be completed. The purpose of these experiments is to demonstrate the utility of the exploratory evolutionary search for event sequences methodology as implemented in the developed tool to inform the decision on a subset of security mechanisms to provide security-in-depth.

This experiment utilizes the full suite of metrics and the complete set of programs as seen in the first experiment in Section 8.1. A detailed XML configuration file for this experiment can be found in the Appendix as Figure A.17, as seen previously in Chapter 7.3 for a search experiment.

In total, six security mechanisms will be used in these experiments. The **Four** security mechanism instance will initially consist of an installation of **Avast** Free Anti-

Virus 18.6.2349, **ClamWin** 0.98.5, **MalwareBytes** 2.1.4.1018, and **Panda** Security 15.1.0. The second subset **Five** will add **AVG** Anti-Virus Free 17.9.3040. The final subset **Six** will add **AdAware** Free Anti-Virus 11.12.945.9202. It should be noted that **AVG** would only install in a passive state due to other security mechanisms already present on the machine. On the other hand, **AdAware** initially installed in passive mode but was converted into active coverage after acknowledging the previous installation of other security mechanisms. The baseline unsecured virtual machine S^0 (AKA **Without**) instance is configured the same as in Subsection 8.1.1.

From each of the three search experiments, the top event sequence found, as ranked by median fitness, will be selected. The process for re-evaluating event sequences seen in the previous experiment will be followed using each of the three subsets of security mechanisms. An example of the re-evaluation configuration file is described in the Appendix as Figure A.18.

At the same time, any examples of emergent misbehaviour, as reported as virtual machine crashes or event sequence incompletion, will also be evaluated to determine if there is evidence of operational incompatibility beyond what was demonstrated at the time of installation.

8.3.2 Experiment Results

This subsection works through the results of the three experiments. First will be summaries of the search progress, and then information on the metric distributions. This is followed by a description of the top example event sequence from each search experiment, as well as the comparative results of the re-evaluation of each of these example event sequences against all three subsets of security mechanisms. Finally, the top incomplete/crashed event sequences examples produced during the three experiments will be discussed briefly, but fully explored in the final experimental section. Box and whisker plots, as previously described in Subsection 8.2.2, are used to visualize and

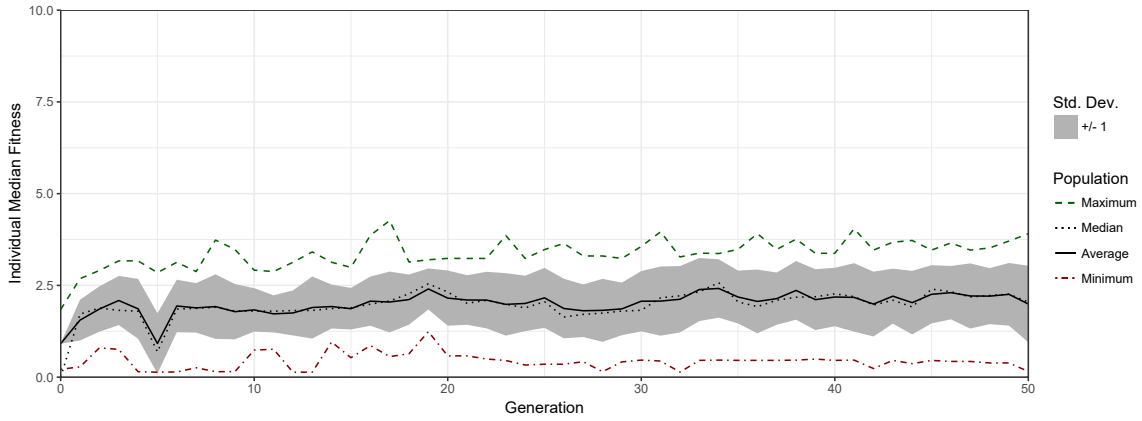
compare distributions.

Each genetic algorithm search process was completed for a population size of twenty-five and fifty generations. The search time, using a single virtual machine individual evaluation thread, for each larger subset size ranged from 2.14, 3.08, and finally 4.58 days. Each re-evaluation of an example event sequence for a single virtual machine instance required about half an hour.

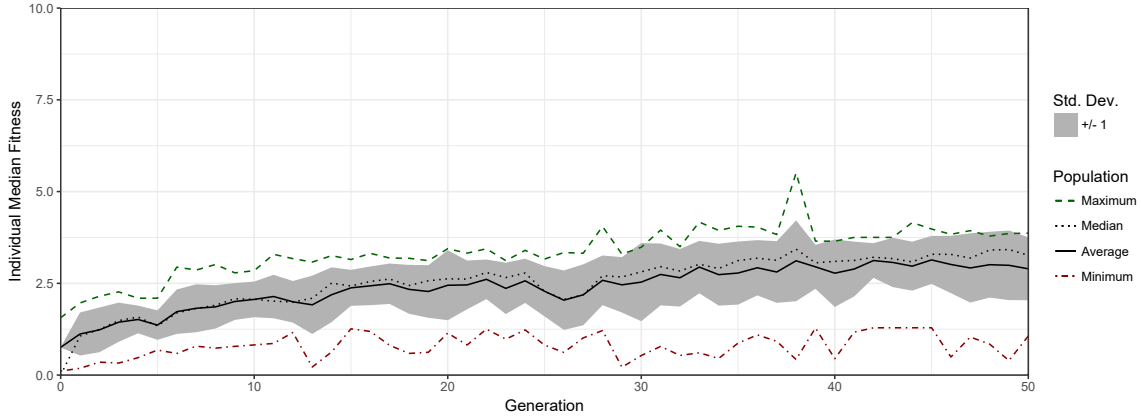
To visualize the population at each generation, Figure 8.25 shows population measures for each generation of the fifty-generation search. Each addition of another security mechanism, to the subset being tested, results in larger fitness values during the process of the search. In particular, the final addition of **AdAware**, a more robust anti-virus suite like **Avast**, to create the subset of **Six** security mechanisms leads to a larger impact to the median fitness. In comparison, the addition of **AVG** in passive mode to create the subset of **Five** security mechanisms has only a minor effect.

Script execution time was only one of many metrics used in this experiment. As done previously, script execution will be examined separately before moving on to the full fitness comparison, which necessarily combines all the metrics. Figure 8.26 reports the sample script execution time measures in minutes. A more descriptive table can be found in the Appendix as Table A.12. As before, the same information but relative to the **Without** virtual machine instance can be found in the Appendix in Figure A.19 and Table A.13. The mean time of subset **Four** is 201.8%, while subset **Five** is at 223.5%, and subset **Six** at 286.5%.

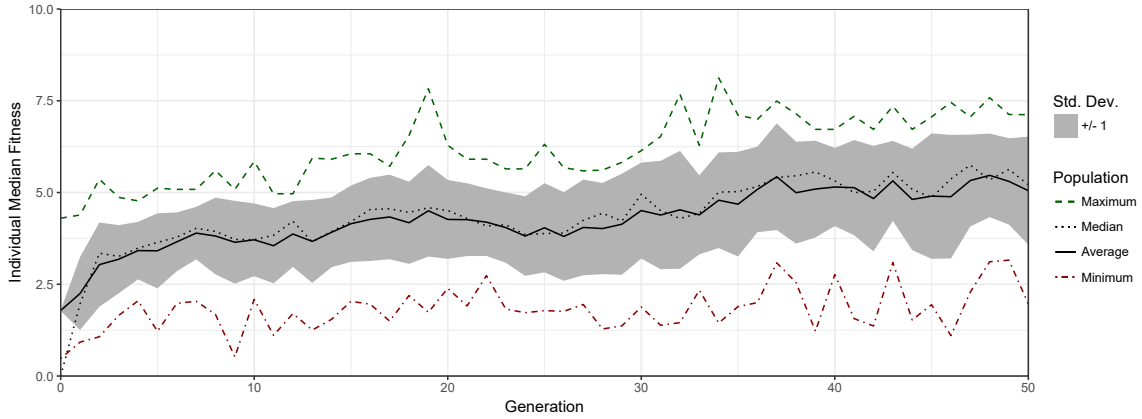
Sample metric values for the full metric suite can be seen in the Appendix in Table A.14. Accompanying the table, from Figure A.20 to Figure A.23 in the Appendix, are visualizations of these sample distributions. These distributions for the full suite of metrics are used to create the fitness value comparisons for the example event



(a) Subset of Four Security Mechanisms



(b) Subset of Five Security Mechanisms



(c) Subset of Six Security Mechanisms

Figure 8.25: Subsets: Population Median Fitness

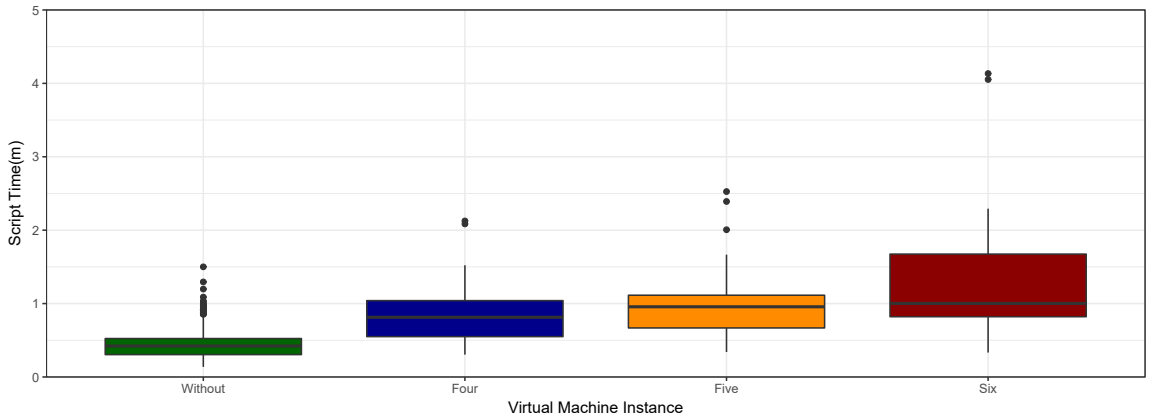


Figure 8.26: Subsets: Sample m^{time} Statistics in m

sequences.

From each of the three search experiments only the top median fitness individual is selected for comparison. For greater detail Table 8.3 shows the top five from each. There were two examples of incompletion from each of the three experiments. These examples are included at the bottom of the table for future reference. For the completed event sequences, for the subsets of **Four** and **Five** mechanisms, there doesn't appear to be any pattern outside of variety of programs being a common feature. For the subset of **Six**, file copying actions, compile actions, and zip actions seem to show a pattern of heavy file access.

Figure 8.27 compares the top event sequence from search experiment on the basis of time. There is no guarantee that time is the portion of full metric suite that the fitness function will maximize. However, the unsurprising slowdown caused by the increase in number of installed security mechanisms can be seen. In particular, the final addition of **AdAware** for subset **Six** has an obvious impact in slowing down the six_1 event sequence and a noticeable impact on $five_1$. The table form of this figure can be found in the Appendix as Table A.15.

A **Without** relative version of Figure 8.27 is depicted in Figure 8.28 and in the Appendix in Table A.16. This comparison normalizes the relative length of script

Table 8.3: Subsets: Event Sequence Examples

event_seq	ev_1	ev_2	ev_3	ev_4	ev_5	ev_6	ev_7	ev_8	ev_9	ev_{10}
$four_1$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Crypt-2	Excel-3	Excel-7	Comp-0
$four_2$	Copy-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Crypt-2	Excel-18	Excel-7	Comp-0
$four_3$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Crypt-2	SQL-0	Excel-7	Comp-0
$four_4$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Crypt-2	Excel-18	Comp-1	Comp-0
$four_5$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Crypt-2	Excel-18	SQL-0	Comp-0
$five_1$	Word-11	SQL-0	Zip-1	Comp-0	SQL-0	Zip-1	IE-0	FTP-0	Zip-1	Zip-1
$five_2$	Word-11	SQL-0	SQL-0	Comp-0	SQL-0	Zip-1	IE-0	FTP-0	Zip-1	Zip-1
$five_3$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Crypt-2	Excel-18	VLC-0	Comp-0
$five_4$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	FTP-0	Crypt-2	Excel-18	Excel-7	Comp-0
$five_5$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Crypt-2	Excel-18	Excel-7	Comp-0
six_1	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	FTP-0	Excel-5	IE-0
six_2	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	FTP-0	SQL-0	IE-0
six_3	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	FTP-0	Empty-0	IE-0
six_4	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	SQL-0	Excel-5	IE-0
six_5	Word-17	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	FTP-0	SQL-0	VLC-0
$four\_inc_1$	Crypt-7	FTP-0	Copy-0	Copy-0	Zip-0	Crypt-1	VLC-0	Comp-1	SQL-0	SQL-0
$four\_inc_2$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Zip-0	Excel-18	SQL-0	Comp-0
$five\_inc_1$	Crypt-7	FTP-0	Copy-0	Copy-0	Zip-0	Crypt-1	VLC-0	Comp-1	SQL-0	SQL-0
$five\_inc_2$	Word-11	SQL-0	SQL-0	Comp-0	SQL-0	Crypt-3	IE-0	FTP-0	Zip-1	Zip-1
six_inc_1	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	Copy-0	SQL-0	IE-0
six_inc_2	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	FTP-0	Excel-5	Copy-0

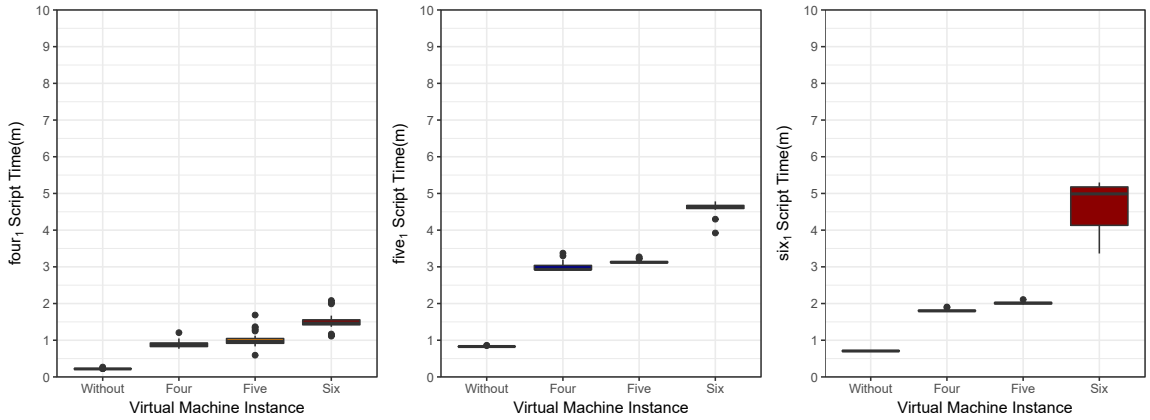


Figure 8.27: Subsets: Event Sequence m^{time} Statistics in m

execution time between the three different example event sequences. Apparent is the consistency in how adding **AVG** in passive mode, to get subset **Five**, has nowhere near the consequence of adding **AdAware** in active mode, to make subset **Six**.

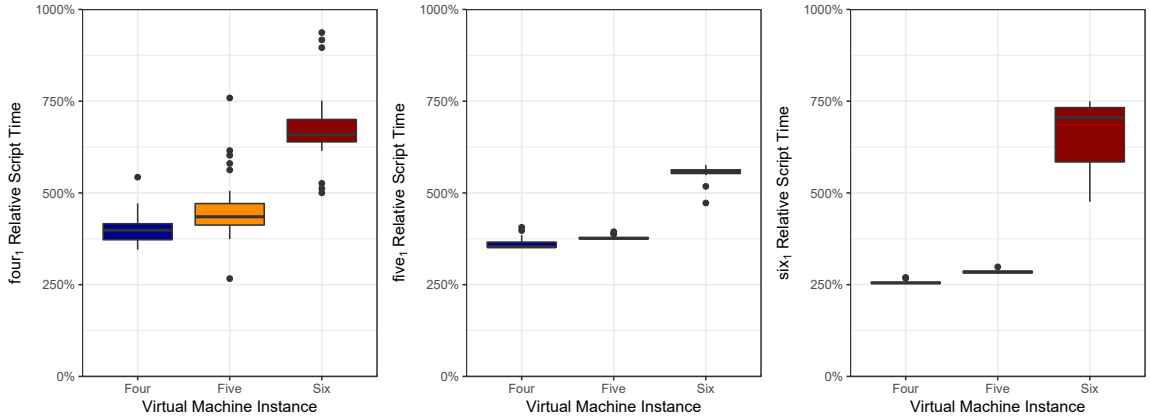


Figure 8.28: Subsets: Event Sequence m^{time} Relative to Without

As repeated previously, comparing event sequence examples based on raw time may be the most grounded metric for an average user. However, the fitness function values are a more accurate quantification of the goal of the search process. This is even more clear in this experiment where a full suite of metrics was used. The event sequence examples being examined were produced as the result of a fitness function merging multiple metrics together to guide the search.

Figure 8.29 shows the comparative comprehensive fitness values for the three ex-

ample event sequences. A table for these values is also in the Appendix as Table A.17. The fitness function comparison creates a red flag for the subset **Six** example, where the event sequence produced extreme fitness values. The example event sequences from the other two search experiments did not produce a unique fitness value between the three subsets of security mechanisms.

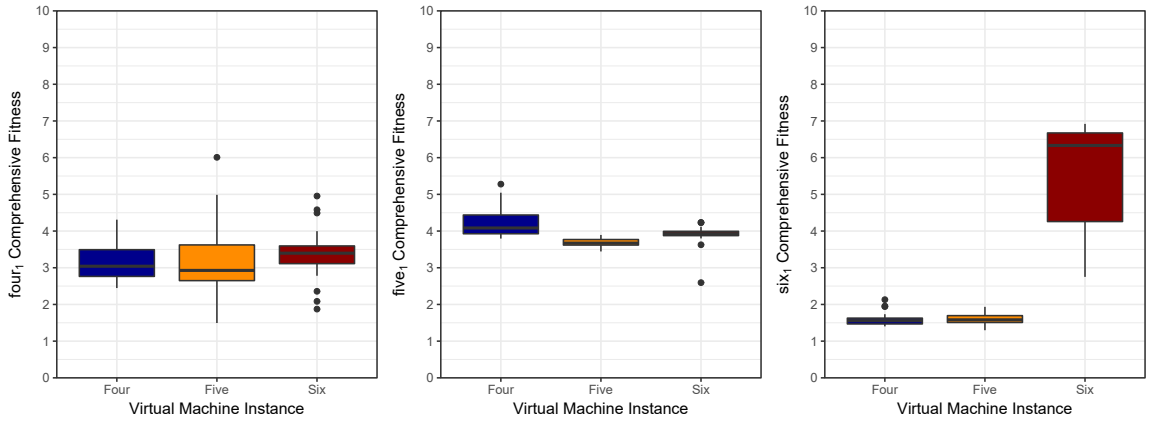


Figure 8.29: Subsets: Event Sequence Comprehensive Fitness

8.3.3 Experiment Discussion

The experimental hypothesis, Hypothesis 8.3.1, was that the results of the developed evolutionary search process could be used to inform the choice between different subsets of multiple security mechanisms for the purpose of determining solutions to the Effective Security-in-Depth problem. The hypothesis was that the search method, described in Chapter 5 and implemented in Chapter 6, could be used to produce challenging example event sequences. These event sequences could then be cross-examined using the exploratory search tool and process in Subsection 7.3 to inform a selection between these subsets of security mechanisms.

Three search experiments were completed using the complete set of programs available and the full suite of security mechanisms. Each search experiment was for an increasingly larger subset of security mechanisms starting at **Four**, which included

Avast/ClamWin/Panda/MalwareBytes, adding **AVG** in passive mode for **Five**, and finally adding **AdAware** in active mode for **Six**. The top event sequence was selected from each, and twenty-five evaluations completed for each of the three subsets of security mechanisms.

If an initial assumption is made that the user has decided that the starting subset **Four** is acceptable, then the reasoning of the process in Subsection 7.3 can be used to decide that the **AVG** addition in passive mode in **Five** has relatively little consequence. However, the addition of **AdAware** in active mode in **Six** has a noticeable performance consequence. The incomplete event sequences, which may be examples of functionality failure that affect this performance decision, will be explored in the following experimental section.

These experiments and evaluations support Hypothesis 8.3.1 that the method, tool, and developed process are suitable for the goal of evaluating Effective Security-in-Depth in regard to performance. The increase to subset **Five** was acceptable, but subset **Six** was not recommended.

Unlike the single security mechanism experiments, these three search experiments were completed without any threading of the search process. This meant only one virtual machine was active at a time. This has a noticeable impact on the length of time required for the search process to complete. On the other hand, this resulted in none of the search experiments having false positive event sequences with errors where the virtual machine failed during startup or shutdown.

Each of the three experiments had two examples of incomplete event sequences. $four\_inc_1$ was incomplete in 2 of 2 executions. $four\_inc_2$ was incomplete in 3 of 28 executions. $five\_inc_1$ was incomplete in 1 of 12 executions. $five\_inc_2$ was incomplete in 3 of 20 executions. six_inc_1 was incomplete in 3 of 4 executions. six_inc_2 was incomplete in 1 of 2 executions. None of these examples are necessarily confirmations

of emergent misbehaviour, but are clearly of interest. All six of these will be examined in the following experimental section.

8.4 Examples of Emergent Misbehaviour

The last hypothesis explored in this thesis involves investigating examples of emergent misbehaviour, such as the examples of functionality failure found during the previously completed exploratory search experiments.

Hypothesis 8.4.1. Verifying Extreme Emergent Misbehaviour
The process described in Section 7.4 is effective at validating and exploring interaction event sequence examples of emergent misbehaviour. This exploratory process uses the software tool implemented in Chapter 6 to re-evaluate event sequences.

This experimental section performs re-evaluation tests of twenty-five executions for each of the nine examples of prospective emergent misbehaviour found in the previous experiments. Each of these examples was an instance of an event sequence not completing execution before a time limit of five minutes was reached. Example event sequences, which continue to be of interest after these re-evaluations, will be further explored with an attempt to create minimal working examples of failure. Each re-evaluation of an example event sequence for a single virtual machine instance ranged from half an hour, for those without incompletions, to two hours, for those with incompletions, due to the requirement of a minimum of five minutes per incomplete re-evaluation.

In addition to the previous nine, two additional examples of emergent misbehaviour found in similar search experiments will be discussed. The first is an example where Web site security certificates were not accepted by a security mechanism. The second more interesting example was found during a similar experiment to that completed in Section 8.3 examining a subset of security mechanisms. However, for this

example, the security mechanisms installed were different versions, and the **AVG** security mechanism was enabled in active instead of passive mode. This example was discovered in a search, validated via re-evaluation, reduced to a more minimal example, and then expanded via a variant of the evolutionary search process to produce a higher rate of failure. This validation process included the use of a physical machine reproduction of the virtual machine instance to verify the example. The configuration of the physical machine was introduced in Subsection 8.1.1.

Table 8.4 repeats the nine event sequence examples found to be incomplete in the previous experiments. Each of these event sequences will be re-evaluated twenty-five times to determine if the incomplete results are a repeatable emergent misbehaviour. An example that is repeatable is of interest for further examination. Each re-evaluation will be completed with all the virtual machine instance configurations from the experiments that produced the example. For instance, the incomplete usage profile examples will be re-evaluated against **Without**, **ClamWin**, **Avast**, and **AdAware**, and the subset examples will be re-evaluated against **Without** along with subsets **Four**, **Five**, and **Six**.

Table 8.4: Incomplete Event Sequence Examples

event_seq	ev_1	ev_2	ev_3	ev_4	ev_5	ev_6	ev_7	ev_8	ev_9	ev_{10}
$avast\_inc_{1,1}$	IE-0	Copy-0	Crypt-5	IE-0	Copy-0	Copy-0	Comp-0	Copy-0	Copy-0	Excel-18
$avast\_inc_{1,2}$	IE-0	Copy-0	Crypt-5	IE-0	IE-0	Crypt-7	Comp-0	Copy-0	Copy-0	Excel-18
$avast\_inc_{2,1}$	Word-3	VLC-0	Word-16	Word-20	SQL-0	Word-20	Word-8	Word-16	Word-3	Zip-1
$four\_inc_1$	Crypt-7	FTP-0	Copy-0	Copy-0	Zip-0	Crypt-1	VLC-0	Comp-1	SQL-0	SQL-0
$four\_inc_2$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Zip-0	Excel-18	SQL-0	Comp-0
$five\_inc_1$	Crypt-7	FTP-0	Copy-0	Copy-0	Zip-0	Crypt-1	VLC-0	Comp-1	SQL-0	SQL-0
$five\_inc_2$	Word-11	SQL-0	SQL-0	Comp-0	SQL-0	Crypt-3	IE-0	FTP-0	Zip-1	Zip-1
six_inc_1	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	Copy-0	SQL-0	IE-0
six_inc_2	Word-3	Copy-0	Copy-0	Copy-0	Comp-0	Zip-1	Copy-0	FTP-0	Excel-5	Copy-0

Table 8.5 contains the incompletion count for the three usage profile examples,

and Table 8.6 shows the average script execution time. In twenty-five re-evaluations for each of the examples, against all four virtual machine instances, there was not a single example of an event sequence being incomplete. There were also no instances of there being an error starting up or shutting down the virtual machine.

Table 8.5: Usage Profile Examples Incompletion Count out of 25

event_seq	*Without*	*ClamWin*	*AdAware*	*Avast*
$avast\_inc_{1,1}$	0	0	0	0
$avast\_inc_{1,2}$	0	0	0	0
$avast\_inc_{2,1}$	0	0	0	0

Table 8.6: Usage Profile Examples m^{time} Statistics in m

event_seq	*Without*	*AdAware*	*Avast*	*ClamWin*
$avast\_inc_{1,1}$	0.56	0.55	0.84	0.66
$avast\_inc_{1,2}$	0.46	0.44	0.67	0.55
$avast\_inc_{2,1}$	0.35	0.35	0.42	0.45

Each of these three examples were only incomplete a single time during their original search process executions. These original incompletions were during a threaded search process where there were a number of virtual machine errors, in which the load on the system disrupted virtual machine start up. These errors, and also the incompletions, disappeared with the single threaded test process completed to re-evaluate these individuals. The lack of repeatability of these incompletions leads to the conclusion that these events may have been from ulterior causes than from the sequence of events itself. As a result, these three examples aren't of clear interest and the performance conclusions for each of the usage profiles developed in the experiments will remain the best evidence for choosing between the security mechanism options.

In general, it can be noted that the methodology developed was designed to account for the possible occurrence of failures that could not be reproduced. The failures were tracked for later examination and the highest rate occurring ones introduced into the genetic pool of the algorithm by genetic operators. Retaining the genetic material of failure examples in the population allows for the possibility that if the behaviour was interesting, then genetic information is still propagated to future generations. However, the failure rate of such incomplete individuals was not used by the fitness function. This was due to the evident danger of it misleading the search process. This is similar to why the median was used for fitness values instead of the maximum, which might have been an outlier as well.

Table 8.7: Subset Examples Incompletion Count out of 25

event_seq	*Without*	*Four*	*Five*	*Six*
$four\_inc_1$	0	14	0	6
$four\_inc_2$	0	1	0	7
$five\_inc_1$	0	0	0	0
$five\_inc_2$	0	0	0	0
six_inc_1	0	0	0	0*
six_inc_2	0	0	0	0*

(*)Only when script time limit set to 10 minutes

Table 8.7 contains the incompletion count for the six examples from the subsets of security mechanism experiments, and Table 8.8 shows the average script execution time. The two event sequences from the **Four** experiment appear to be of interest, producing incomplete results, despite running much shorter than the five-minute limit. In fact, in each example, the failure point is during the first $Zip - 0$ interaction event. The two **Six** examples did produce incompletions, but only for the **Six** virtual machine

instance and only with the time limit of five minutes. In Table 8.8, the incompletion count is actually reported for an increased time limit of ten minutes. With a limit of five minutes, all twenty-five executions were incomplete.

Table 8.8: Subset Examples m^{time} Statistics in m

event_seq	*Without*	*Four*	*Five*	*Six*
$four\_inc_1$	0.30	0.43	0.42	0.59
$four\_inc_2$	0.89	1.29	1.40	1.96
$five\_inc_1$	0.54	1.89	2.09	2.74
$five\_inc_2$	0.70	2.41	2.62	3.41
six_inc_1	0.77	1.84	2.05	5.42
six_inc_2	0.77	1.84	2.05	5.47

From Table 8.7 and Table 8.8, it is possible to determine that only event sequences $four\_inc_1$ and $four\_inc_2$ are of interest for further examination. It has been verified that the emergent misbehaviour was repeatable and not just a function of a script time limit. From this point, the process moves from validation to creating a minimal wokring example of incompletion for each of these event sequences.

Table 8.9 reports the incompletion counts for reduced sub-sequences taken from event sequence $four\_inc_1$, as evaluated against subset **Four**. First, by examining the process of running $four\_inc_1$, it was determined that the event sequence halted at $Zip - 0$ with an error because the zip action was unexpectedly unable to access a file it expected to be able to add to a new zip file. The examination starts with a minimal partial sub-sequence $four\_inc_{1,part}$ that includes the first five program interactions.

The next five smaller sub-sequence examples created from $four\_inc_{1,part}$ are $four\_inc^1_{1,part}$ to $four\_inc^5_{1,part}$. Each is the result of removing ev_1 to ev_5 from $four\_inc_{1,part}$. From these five comes the implication that the first $Crypt - 7$ interaction and the last

Zip − 0 interaction are required. The final three sub-sequence examples are the result of keeping the first and last event in $four\_inc_{1,part}$ but removing different combinations of the three middle events. The result is an evident minimal working example of functionality failure $four\_inc^{2,3}_{1,part}$. This is confirmed by the last two event sub-sequences which demonstrate that a removal of either of the *Crypt − 7* or *Copy − 0* events results in no incompletions.

Table 8.9: Subset Four Incomplete Event Sequence One

event_seq	ev_1	ev_2	ev_3	ev_4	ev_5	*Incomplete*	*Note*
$four\_inc_{1,part}$	Crypt-7	FTP-0	Copy-0	Copy-0	Zip-0	13	Up to point of failure
$four\_inc^{1}_{1,part}$		FTP-0	Copy-0	Copy-0	Zip-0	0	Might need Crypt-7
$four\_inc^{2}_{1,part}$	Crypt-7		Copy-0	Copy-0	Zip-0	22	May not need FTP-0
$four\_inc^{3}_{1,part}$	Crypt-7	FTP-0		Copy-0	Zip-0	22	May not need first Copy-0
$four\_inc^{4}_{1,part}$	Crypt-7	FTP-0	Copy-0		Zip-0	22	May not need second Copy-0
$four\_inc^{5}_{1,part}$	Crypt-7	FTP-0	Copy-0	Copy-0		0	Need Zip-0
$four\_inc^{2,3}_{1,part}$	Crypt-7			Copy-0	Zip-0	22	Don't need FTP-0 and one of Copy-0
$four\_inc^{3,4}_{1,part}$	Crypt-7	FTP-0			Zip-0	0	Need at least one Copy-0
$four\_inc^{2,3,4}_{1,part}$	Crypt-7				Zip-0	0	Need at least one of middle events
$four\_inc^{1,2,3}_{1,part}$				Copy-0	Zip-0	0	Need Crypt-7
$four\_inc^{2,3,4}_{1,part}$	Crypt-7				Zip-0	0	Need Copy-0

Table 8.10 reports the incompletion counts for reduced sub-sequences taken from event sequence $four\_inc_2$ as evaluated against subset **Four**. Again, $four\_inc_2$ was determined to halt at the first *Zip − 0* with the same error as before. The examination starts with a minimal partial sub-sequence $four\_inc_{2,part}$ that includes the first seven program interactions.

The next seven smaller sub-sequence examples created from $four\_inc_{2,part}$ are $four\_inc^{1}_{2,part}$ to $four\_inc^{7}_{2,part}$. Each is the result of removing one of the seven events. Given the low occurrence of incompletions, it is less clear which of these single event removals has positive or negative consequences. Attempting to remove all four single

events that had no incompletions in the next sub-sequence $four\_inc_{2,part}^{1,2,5,6}$ resulted in no incompletions. As a result, in $four\_inc_{2,part}^{1,2}$, only the first two events were removed.

Since the $four\_inc_{2,part}^{1,2}$ sub-sequence produced incompletions, the next step was to remove each of four events prior to $Zip-0$. The result of this was an implication that $SQL-0$ and/or $VLC-0$ were not required. The next event sub-sequence $four\_inc_{2,part}^{1,2,3,6}$ removed both of these and produced incompletions. The final two sub-sequences removed one of each of the two leading events in the triple but neither sub-sequence produced incompletions. The result is an evident minimal working example of functionality failure $four\_inc_{2,part}^{1,2,3,6}$.

Table 8.10: Subset Four Incomplete Event Sequence Two

event_seq	ev_1	ev_2	ev_3	ev_4	ev_5	ev_6	ev_7	*Incomplete*	*Note*
$four\_inc_{2,part}$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0	Zip-0	1	Up to point of failure
$four\_inc_{2,part}^{1}$		IE-0	SQL-0	Word-16	FTP-0	VLC-0	Zip-0	1	May not need First SQL-0
$four\_inc_{2,part}^{2}$	SQL-0		SQL-0	Word-16	FTP-0	VLC-0	Zip-0	1	May not need IE-0
$four\_inc_{2,part}^{3}$	SQL-0	IE-0		Word-16	FTP-0	VLC-0	Zip-0	0	Might need second SQL-0
$four\_inc_{2,part}^{4}$	SQL-0	IE-0	SQL-0		FTP-0	VLC-0	Zip-0	0	Might need Word-16
$four\_inc_{2,part}^{5}$	SQL-0	IE-0	SQL-0	Word-16		VLC-0	Zip-0	2	May not need FTP-0
$four\_inc_{2,part}^{6}$	SQL-0	IE-0	SQL-0	Word-16	FTP-0		Zip-0	3	May not need VLC-0
$four\_inc_{2,part}^{7}$	SQL-0	IE-0	SQL-0	Word-16	FTP-0	VLC-0		0	Need Zip-0
$four\_inc_{2,part}^{1,2,5,6}$			SQL-0	Word-16			Zip-0	0	Might need one of four removed
$four\_inc_{2,part}^{1,2}$			SQL-0	Word-16	FTP-0	VLC-0	Zip-0	1	May not need first SQL-0 and IE-0
$four\_inc_{2,part}^{1,2,3}$				Word-16	FTP-0	VLC-0	Zip-0	2	May not need second SQL-0
$four\_inc_{2,part}^{1,2,4}$			SQL-0		FTP-0	VLC-0	Zip-0	0	Might need Word-16
$four\_inc_{2,part}^{1,2,5}$			SQL-0	Word-16		VLC-0	Zip-0	0	Might need FTP-0
$four\_inc_{2,part}^{1,2,6}$			SQL-0	Word-16	FTP-0		Zip-0	1	May not need VLC-0
$four\_inc_{2,part}^{1,2,3,6}$				Word-16	FTP-0		Zip-0	2	May not need second SQL-0 and VLC-0
$four\_inc_{2,part}^{1,2,3,4,6}$					FTP-0		Zip-0	0	Might need Word-16
$four\_inc_{2,part}^{1,2,3,5,6}$				Word-16			Zip-0	0	Might need FTP-0

After the effort in producing minimal working sub-sequence examples, Table 8.11 re-evaluates these minimal examples against all three original subsets of security mechanisms **Four**, **Five**, and **Six**. The incompletion rates of both original examples

were increased in subset **Four** and now occur in all three subsets.

Table 8.11: Minimal Examples Incompletion Count out of 25

event_seq	*Without*	*Four*	*Five*	*Six*
$four\_inc^{2,3}_{1,part}$	0	21	5	5
$four\_inc^{1,2,3,6}_{2,part}$	0	2	4	1

The hypothesis of why the larger subsets have fewer occurrences of incompletion is that the longer delays found during the execution of the same sequence of actions on a larger subset allows the file handle to become accessible in time. The issue still occurs in these larger subsets, so the emergent behaviour exists in all the subsets, but the timing of actions that cause its occurrence is different.

In an effort to determine if there is a minimal set of security mechanisms in which the examples occur, the first minimal event sub-sequence $four\_inc^{2,3}_{1,part}$ was re-evaluated against successively smaller subsets of security mechanisms created by removing the more passive and lighter weight security mechanisms. The first step was to remove **ClamWin** to produce subset **Three**. The second step was to remove **MalwareBytes** to produce subset **Two**. Then the event sub-sequence was run against **Avast** and **Panda** as individual security mechanisms configurations. Table 8.12 reports these results. It appears that the combination of **Avast** and **Panda** was necessary to produce consistent incompletion errors.

The hypothesis for this emergent misbehaviour is that the monitoring processes of the combined **Avast** and **Panda**, when delayed by the prior two events, results in the Zip action being unable to get proper file handle access during the Zip process, creating a generic Windows XP "*Access Denied*" pop-up box. This pop-up box is produced whenever Windows XP determines access to a requested file was denied due to permissions or the file being in use. The prior two actions do not make use of any

file being used in the Zip action. The directory being zipped is static and no files have moved. A common forum posting solution to this type of unexplainable behaviour in Windows XP recommends disabling anti-virus software due to its disruption to file handle access and load delays with file movement. Microsoft support pages for Windows XP addressing this reported issue do not appear to be available anymore, due to Windows XP forum support being discontinued.

Table 8.12: Minimal Examples Incompletion Count out of 25

event_seq	*Three*	*Two*	*Avast*	*Panda*
$four\_inc^{2,3}_{1,part}$	19	18	0	0

The final two examples of emergent misbehaviour were not produced during the previous experiments discussed in this thesis. Both examples were produced during the development of the methodology. The first example is one where the observed behaviour was resulting in an incomplete event sequence. The behaviour was a configuration issue where one of the security mechanisms was interacting with the browser result when navigating to a specific Web site address. The second example contains incompletion behaviour during a $Zip - 0$ action as seen previously, but found for different version configuration of security mechanisms.

The experiments that produced these examples were being performed when the initial metric selection and parameter settings were being developed. One challenge of security mechanism research has been that many mechanisms will pull setup files from the Internet. Between this earlier stage and the experiments previously reported, the Windows 10 host machine underwent an update that broke the operation of the version of Virtual Box installed at that time, which also broke access to the contained virtual machine instance configurations. The attempt to recreate the virtual machine instances had to be completed on a newer version of Virtual Box than what was used

for these two earlier examples. This recreation resulted in different versions of some of the security mechanisms. In some cases, offline installation files allow for this to be controlled, but in many cases security mechanism creators prefer not to offer offline installation options for older versions in any capacity. Although security mechanism have been updated, some behaviour, such as the failure associated with the $Zip - 0$ action, is not always repaired.

It has been shown previously that not every incompletion is of interest. For example, the event sequence examples produced for subset **Six** were determined to just be event sequences with a run time longer than the limit that had been set on the search process. Similarly, incomplete event sequences need to be visualized and the script execution tracked to determine if event sequence incompletions are of interest or if other variables are in play.

During the experiments earlier in the methods development, there was another instance of the developed search methodology finding an example that originally appeared interesting, but was determined to be from other causes. In this example, when the search completed, the list of incomplete examples was examined. The individual at the top with the highest rate of incompletions was selected. This individual did not finish execution three out of three times. This event sequence consisted of five events followed by a Web navigation, a video start event, and another Web navigation event. The Web navigation event directed Internet Explorer to a Web page with a security certificate not accepted by one of the installed security mechanisms.

This individual was re-simulated 100 times for the **Without** system and showed completion problems in 5% of tests, and in 100% of test with the secured system. As a result, the script execution was visualized. It was determined that its execution was disrupted by the two Web navigation events. Execution would eventually complete, but well after the five-minute expiration time for script execution. Further examina-

tion revealed that this execution delay could be reduced by changing the time of the virtual machine instance to make the security certificate of the Web site valid again. The security mechanism disruption had been the result of a security certificate that was not longer valid in regard to the virtual machine internal system time.

To this point, the process of verifying the existence of an example of incompletion and determining a minimal working sub-sequence has been validated. There are two stages remaining. The first stage is to verify the behaviour in a physical system to determine that the virtual machine simulation is not the cause of the incompletion. The second, and possibly optional, stage is determining if the rate of incompletion can be increased by the addition of interaction events to the minimal working example.

Another similar experiment, but for different versions of the anti-virus programs[1], produced a minimal example event sequence consisting of doing a zip folder compression $Zip - 0$, a 3DES encryption action $Crypt - 5$, and another zip folder compression $Zip - 0$. This example was 90% repeatable in the virtual machine configuration and was reproduced at the same rate on a physical machine reproduction of the virtual machine. This step of validating an example against a physical machine provides confirmation that the behaviour is not just a result of virtual machine emulation.

In addition to a minimal failing example, it is sometimes desirable to provide the creator of a security mechanism with as consistent an example as possible of a certain behaviour. To accomplish this goal, the developed search methodology can be used through a simple modification. This modification still includes performing an exploratory event sequence search. However, every event sequence is treated as a prefix to a static event sequence provided at the beginning of the search. For example, the three-event sequence found previously was treated as this static event sequence for such a search. For this exploratory search, the prefix event sequence of length two

[1]Avast Free Anti-Virus 17.1.3388, ClamWin 0.98.5, Panda Security 16.1.1, MalwareBytes 2.1.4.1018, Ad-Aware Free Anti-Virus 11.8.586.8535 (Limited Compatibility Mode), and AVG Anti-Virus Free 16.161.8048

is the only part modified during the search process. The search goal was to find a subsequence of length five with a higher rate of incompletion. The result of this search was an event sequence that added a $Zip - 0$ action followed by a $Comp - 0$ action as a prefix. These additional interaction events were able to push the repeatability to 98% on the physical machine.

The experimental hypothesis, Hypothesis 8.4.1, was that the results of the developed process could be used to validate and explore prospective examples of emergent misbehaviour. Such examples include functionality failure, where an event sequence fails to complete. The experimental evaluations completed in this final section demonstrated that prospective incomplete event sequences could be filtered for those that were repeatable using re-evaluation tests. These event sequences could then be reduced to minimal working examples. These minimal working examples could then be validated on a physical machine configuration of the virtual instance.

The last thing demonstrated in this section was that minimal examples could then be expanded to produce examples with a higher rate of incompletion. This additional process is not necessarily of interest to the regular user. It is primarily of interest to a system administrator or the developer of one of the security mechanisms. The combination of a minimal working example and a highly repeatable failure example, will allow for the issue to be more concretely identified and fixed, or ignored with precise statements about compatibility. As the evaluations have shown, there are security system combinations that are more compatible and these combinations are the most suitable options to achieve Effective Security-in-Depth.

Chapter 9

Conclusions and Future Work

When examining information technology systems, there exists the important challenge of securing them against undesirable exploitation and penetration. Technology-wise, after the operating system, the choice of security mechanism(s) is one of the most important in protecting a computer system. There are a wide variety of choices of which security mechanism(s) to choose, as each provides a different type and range of coverage. Each organization and/or user can be expected to have different operational requirements for their system's performance. Balancing these requirements, while maintaining security, is a key challenge for procurement and deployment of information security assets.

Different individual users will have different operational requirements, depending on what programs they install, what interactions they have with those programs, and what level of performance they expect from their computing system. At the same time, each security mechanism has different resource requirements and operational performance consequences. A user may find particular security mechanisms disruptive based on their usage profile, while another user may not. It is a challenging problem to identify and select between security mechanisms, given the large possible range and ordering of program interactions.

Security mechanisms provide coverage against system penetration and exploitation of vulnerabilities. The pursuit of security necessarily must balance the other operational requirements of the secured system. The ideal secured system would consist of a multitude of security mechanisms for each area of coverage. However, too many security mechanisms will likely result in competition for operating resources

and control of monitoring points. Security-in-Depth uses layered defence of multiple security mechanisms to provide redundancy to the coverage of the secured system, and to increase the investment necessary for the attacker to achieve success.

Ideally, security mechanisms would co-operate to make use of all their strengths, while covering for each of their weaknesses. Business, research, and software development goals generally create an environment where mechanisms are incompatible with competitors and have conflicts for system resources and monitoring points. Many security mechanisms advise against, or even prevent, simultaneous deployment and operation of other vendor's software. Emergent behaviour is a further challenge to the problem of Security-in-Depth. Emergent behaviour is unpredictable behaviour that emerges from the interaction of multiple entities, such as a collection of security mechanisms fighting for control over resources. Given that collaboration is not a common design goal, inter-operation is likely to result in negative functional or performance consequences that cannot be discovered in the design phase or isolated operation.

The Effective Security-in-Depth problem is the determination of a subset of security mechanisms that maximize the system's functionality and performance while maintaining, or exceeding, a desired level of security coverage. To assist the selection of security mechanisms to protect a system, this thesis first formalized the problem of Effective Security-in-Depth, and then targeted its challenges.

The first challenge was that of measuring the performance of a system in regard to all the variations of user interactions possible for a specific user profile and across a variety of metrics. Overcoming this challenge was accomplished through the use of a genetic algorithm search of interaction event sequences designed to integrate a variety of different metrics into a single fitness function. The second challenge was that of the stochastic nature of system operation and the resultant variations in performance

which produce a distribution of values for each metric. This variation also includes outliers that may be misleading distractions to the search goal. Overcoming this challenge was accomplished through the use of initial sample metric distributions, the collection of multiple fitness evaluations for an individual during a search, and the use of median fitness values instead of extreme outliers.

The combination of these challenge answers formed a methodology designed to find solutions to the Effective Security-in-Depth problem. This methodology was implemented as a software tool capable of evaluating interaction event sequences in a virtual machine environment. This tool was then supplemented with three descriptive processes that utilized the tool to choose between singular security mechanisms, compare subsets of multiple security mechanisms, and explore examples of functionality failure. The goal was a design that integrated usage profiles and allowed for the automated exploration of possible interaction event sequences to produce candidate examples that allowed for different prospective security mechanism(s) to be compared and contrasted.

In order to demonstrate the capabilities of the developed methodology, the progress of representative evolutionary search executions was explored. The experimental evaluation of the search process demonstrated that the fitness function was successful in directing the evolutionary search method's progress. This successful progress was made despite the challenges of dynamic simulation noise in determining a point in the problem's search space. Next, experiments were completed utilizing each of the three proposed processes using the tool. The experimental evaluation of the three processes demonstrated the capability of the software tool to examine performance costs, contrast and select between different security suites, and verify examples of emergent misbehaviour observed.

This thesis is only an initial foray into this less explored area of study. There

are many different opportunities for future developments. There is always the potential for further improvements to any existing system by simply addressing the same problem, but with different requirements, or diverging from current research with a novel idea or application. The following summarizes a selection of possible routes of continuing research for the existing work, as well as tangential considerations.

This thesis relies on possibly a too simplistic view of usage profiles. It is possible that usage profile data could be considered as a distribution of behaviour, such that certain actions can be expected to occur more/less often. This could be addressed in numerous ways. It is possible that, like a recommendation system, information could be collected from a user's system instead of explicitly determined as it was in this thesis. It is also possible that, during the evolutionary search, event sequences that are too extreme relative to a user's usage profile, for example too many repeating similar actions, might be avoided through genetic algorithm methods. This would require treating the user's usage profile as a more complex distribution of interaction events. However, there is the danger that the most critical interaction event sequences might be uncommon, but still have the most interesting and relevant behaviour.

The evolutionary search process could be improved with a variety of different genetic algorithm research ideas. One of the most relevant was hinted at during the final experiment where, in the process of evaluating a misbehaving individual, a critical sub-sequence was used as a suffix to all event sequences in an additional evolutionary search. The purpose of this modified search was to produce a sequence with greater misbehaviour. As a similar idea, it might be possible to kick-start future searches by seeding the initial population of a search with such minimal examples of emergent misbehaviour.

Along the same lines, it is possible that particular event sequences will be found that produce behaviour that is already known and desirable to ignore in a search with

a goal to find other unique event sequences. A heavy-handed method would be to remove the involved programs from future searches. On the other hand, there are a number of interesting prospective ways of addressing this issue by ignoring certain sub-sequences. This may be accomplished by either not allowing them to be created, or penalizing their fitness values during evaluation.

In this thesis, only the singular class of anti-virus suite security mechanisms was evaluated. There are a number of different areas of security protection that provide unique requirements to research, such as focusing on intrusion detection system inter-operation. Similarly, heterogeneous security mechanisms might also inter-operate differently. It should be noted that many anti-virus suites are often heterogeneous collections of security mechanisms. Similarly, only one operating system was explored. It is possible that the environment of a non-Windows architecture, or even a more modern Windows architecture beyond Windows XP, may present a different security monitoring environment. This environment could possibly be less disruptive to inter-operation or security mechanisms' performance.

Lastly, the key developments made for the methodology were designed to attack problems where the fitness landscape of a solution was best represented by a distribution, rather than a single value. Particularly, the method attacked the danger of untrustworthy outliers and merging multiple metrics with varying distributions into a single fitness function. There are possibly other problems that share these challenging multi-objective characteristics for which parts of this method may assist in providing a solution.

Bibliography

[ASHBH05] Ehab Al-Shaer, Hazem Hamed, Raouf Boutaba, and Masum Hasan. Conflict Classification and Analysis of Distributed Firewall Policies. *IEEE Journal on Selected Areas in Communications*, 23(10):2069–2084, 2005.

[BZHH11] Carlos Bacquet, A. Nur Zincir-Heywood, and Malcom I. Heywood. Genetic Optimization and Hierarchical Clustering Applied to Encrypted Traffic Identification. In *IEEE Symposium on Computational Intelligence in Cyber Security*, CICS'11, pages 194–201, April 2011.

[BAP13] BAPCo. SYSMark. `http://bapco.com/products/sysmark-2012`, 2013. [Online; accessed 1-September-2013].

[Bel06] Steven M. Bellovin. On the Brittleness of Software and the Infeasibility of Security Metrics. *IEEE Security & Privacy*, 4:96–96, 2006.

[BD13] Karel P. Bergmann and Jörg Denzinger. Testing of Precision Agricultural Networks for Adversary-Induced Problems. In *Conference on Genetic and Evolutionary Computation Conference*, GECCO'13, pages 1421–1428, 2013.

[BFL96] Matt Blaze, Joan Feigenbaum, and John O. Lacy. Decentralized Trust Management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[CBJW09] Yin Chao, Cao Bingyao, Ding Jiaying, and Gu Wei. The Research and Implementation of UTM. In *International Communication Conference on Wireless Mobile and Computing*, CCWMC'09, pages 389–392, 2009.

[Coh13] Cindy Cohn. The NSA is Making Us All Less Safe. `https://www.eff.org/deeplinks/2013/10/nsa-making-us-less-safe`, 2013. [Online; accessed 1-September-2013].

[CCC+05] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. *Operating Systems Review*, 39:133–147, 2005.

[DJ75] Kenneth Alan De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[DLY+08] Fachao Deng, An'an Luo, Zhang Yaokun, Zhen Chen, Xuehai Peng, Xin Jiang, and Dongsheng Peng. TNC-UTM: A Holistic Solution to Secure Enterprise Networks. In *International Conference for Young Computer Scientists*, ICYCS'08, pages 2240–2245, 2008.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[DFF97] Jörg Denzinger, Marc Fuchs, and Matthias Fuchs. High Performance ATP Systems by Combining Several AI Methods. In *Proceedings of the 15th International Joint Conference on Artifical Intelligence*, volume 1 of *IJCAI'97*, pages 102–107, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 26:29–41, 1996.

[FTD09] Tom Flanagan, Chris Thornton, and Jörg Denzinger. Testing Harbour

Patrol and Interception Policies using Particle-Swarm-Based Learning of Cooperative Behavior. In *Proceedings of the Computational Intelligence for Security and Defense Applications*, CISDA'09, pages 1–8, 2009.

[Fut13] Futuremark. PCMark. `http://www.futuremark.com/benchmarks/pcmark`, 2013. [Online; accessed 1-September-2013].

[GL03] Fred Glover and Manuel Laguna. Tabu Search. *Inteligencia Artificial, revista Iberoamericana De Inteligencia Artificial*, 7:29–48, 2003.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Gol11] Dieter Gollmann. *Computer Security*. Wiley, 3rd edition, 2011.

[GFX12] Mark Grechanik, Chen Fu, and Qing Xie. Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing. In *International Conference on Software Engineering*, ICSE'12, pages 156–166, 2012.

[HH98] Randy Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley & Sons Inc., 1998.

[Hol75] John Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[Hol98] John Holland. *Emergence: From Chaos to Order*. Addison-Wesley, 1998.

[HY08] Xue Hong-Ye. MultiCore Systems Architecture Design and Implementation of UTM. In *International Symposium on Information Science and Engineering*, ISISE'08, pages 441–445, 2008.

[HPW05] Michael Howard, Jon Pincus, and Jeannette M. Wing. Measuring Relative Attack Surfaces. In D.T. Lee, S.P. Shieh, and J.D. Tygar, editors, *Computer Security in the 21st Century*, pages 109–137. Springer US, 2005.

[HZP09] Nan Hu, Jie Zhang, and Paul A. Pavlou. Overcoming the J-shaped Distribution of Product Reviews. *Commun. ACM*, 52(10):144–147, October 2009.

[Hud11] Jonathan Hudson. Risk Assessment and Management for Efficient Self-Adapting Self-Organizing Emergent Multi-Agent Systemss. Master's thesis, University of Calgary, 2011.

[HD15] Jonathan Hudson and Jörg Denzinger. Risk Management for Self-Adapting Self-Organizing Emergent Multi-Agent Systems Performing Dynamic Task Fulfillment. *Autonomous Agents and Multi-Agent Systems*, 29(5):973–1022, 2015.

[HDKB10] Jonathan Hudson, Jörg Denzinger, Holger Kasinger, and Bernhard Bauer. Efficiency Testing of Self-Adapting Systems by Learning of Event Sequences. In *Proceedings of the International Conference on Adaptive and Self-adaptive Systems and Applications*, ADAPTIVE '10, pages 200–205, 2010.

[HDKB11] Jonathan Hudson, Jörg Denzinger, Holger Kasinger, and Bernhard Bauer. Dependable Risk-Aware Efficiency Improvement for Self-Organizing Emergent Systems. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'11, pages 11–20, 2011.

[HMF12] William Hurst, Madjid Merabti, and Paul Fergus. Operational Support for Critical Infrastructure Security. In *International Conference on Embedded Software and Systems*, ICESS'12, pages 1473–1478, 2012.

[KFN93] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*. International Thomas Computer Press, 1993.

[KE95] James N. Kennedy and Russell C. Eberhart. Particle Swarm Optimization. In *International Symposium on Neural Networks*, pages 1942–1948, 1995.

[KK04] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[KVG83] S. Kirkpatrick, M. P. Vecchi, and C. D. Gelatt. Optimization by Simulated Annealing. In *IBM Germany Scientific Symposium Series*, volume 220, pages 671–680, 1983.

[LBS09] Michael E. Locasto, Sergey Bratus, and Brian Schulte. Bickering In-Depth: Rethinking the Composition of Competing Security Systems. *IEEE Security & Privacy*, 7:77–81, 2009.

[LGB10] Michael E. Locasto, Steven J. Greenwald, and Sergey Bratus. Trust Distribution Diagrams: Theory and Applications. In *Proceedings of the Layered Assurance Workshop*, LAW'10, 2010.

[LSK06] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *Network and Distributed System Security Symposium*, pages 95–106, 2006.

[Lon11] Benjamin W. Long. Security in Depth through Smart Space Cascades. In *Australasian Conference on Information Security and Privacy*, ACISP'11, pages 226–240, 2011.

[MCBH10] Bo Ma, Bin Chen, Xiaoying Bai, and Junfei Huang. Design of BDI Agent for Adaptive Performance Testing of Web Services. In *International Conference on Quality Software*, QSIC'10, pages 435–440, 2010.

[MBA09] Sourour Meharouech, Adel Bouhoula, and Tarek Abbes. Ensuring Security in Depth Based on Heterogeneous Network Security Technologies. *International Journal of Information Security*, 8:233–246, 2009.

[MPB+16] Hiren K Mewada, Sanjay Patel, CAP Boyce, A. Nur Zincir-Heywood, MM Sebring, E Shellhouse, ME Hanna, RA Whitehurst, C Dowell, P Ramsted, et al. Artificial Intelligence in Network Intrusion Detection. *Research Journal of Information Technology*, 9(1):74–81, 2016.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[Mog06] Jeffrey C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.

[MHL94] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network Intrusion Detection. *IEEE Network*, 8:26–41, 1994.

[Nat12] National Security Agency. Defense in Depth. `http://www.nsa.gov/ia/_files/support/defenseindepth.pdf`, 2012. [Online; accessed 1-September-2013].

[NPT+09] Cu D. Nguyen, Anna Perini, Paolo Tonella, Simon Miles, Mark Harman, and Michael Luck. Evolutionary Testing of Autonomous Software

Agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, AAMAS'09, pages 521–528. ACM Press, 2009.

[NIS11] NIST/SEMATECH. e-Handbook of Statistical Methods. `http://www.itl.nist.gov/div898/handbook/`, 2011.

[Per11] Chad Perrin. Understanding Layered Security and Defense in Depth. `http://www.techrepublic.com/blog/it-security/understanding-layered-security-and-defense-in-depth/`, 2011. [Online; accessed 1-September-2013].

[QYXL07] Yaxuan Qi, Baohua Yang, Bo Xu, and Jun Li. Towards System-level Optimization for High Performance Unified Threat Management. In *International Conference on Networking and Services*, ICNS'07, pages 7–7, 2007.

[Ros08] Matthew Rosenquist. Defense in Depth Strategy Optimizes Security. Technical report, Intel, September 2008.

[SS94] R.D. Sawyer and M.C. Sawyer. *Sun-Tzu: The Art of War*. Westview Press, 1994.

[SLBK05] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *USENIX Technical Conference*, ATEC'05, pages 149–161, 2005.

[SB14] Mario Silic and Andrea Back. Shadow IT - A View from Behind the Curtain. *Computer Security*, 45:274–283, September 2014.

[Sky06] Skywing. What Were They Thinking: Antivirus Software Gone Wrong. *Uninformed*, 4, June 2006. [Online; accessed 1-September-2013].

[Sma11] P. E. Small. Defense in Depth: An Impractical Strategy for a Cyber World, 2011. [Online; accessed 1-September-2013].

[Smi03] Clifton L Smith. Understanding Concepts in the Defence in Depth Strategy. In *International Carnahan Conference on Security Technology*, pages 8–16, 2003.

[SBE11] Sal Stolfo, Steven M. Bellovin, and David Evans. Measuring Security. *IEEE Security & Privacy*, 9:60–65, 2011.

[Sty04] Martin R. Stytz. Considering Defense in Depth for Software Applications. *IEEE Security & Privacy*, 2:72–75, 2004.

[Sua03] G. Suarez. Challenges Affecting a Defense-in-Depth Security Architected Network by Allowing Operations of Wireless Access Points (WAPs). In *Symposium on Applications and the Internet*, pages 363–367, 2003.

[TFB10] Edward B. Talbot, Deborah A. Frincke, and Matt Bishop. Demythifying Cybersecurity. *IEEE Security & Privacy*, 8:56–59, 2010.

[Thi98] Dirk Thierens. Selection Schemes, Elitist Recombination, and Selection Intensity. In *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 152–159. Morgan Kaufmann, 1998.

[TFY03] Zhi-Hong Tian, Bin-Xing Fang, and Xiao-Chun Yun. An Architecture for Intrusion Detection using Honey Pot. In *International Conference on Machine Learning and Cybernetics*, pages 2096–2100, 2003.

[TNL+07] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Xiaodong Song. Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms. In *EuroSys Conference*, pages 115–128, 2007.

[Val13] Valve Corporation. Programs Which May Interfere with Steam. https://support.steampowered.com/kb_article.php?ref=9828-SFLZ-9289, 2013. [Online; accessed 1-September-2013].

[Wal17] W. H. Waldron. *Elements of Trench Warfare*. National Capital Press, Inc., 1917.

[YMS+06] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen nee Chuah, and Prasant Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *IEEE Symposium on Security and Privacy*, pages 199–213, 2006.

[ZDC+10] Ying Zhang, Fachao Deng, Zhen Chen, Yibo Xue, and Chuang Lin. UTM-CM: A Practical Control Mechanism Solution for UTM System. In *International Conference on Communications and Mobile Computing*, CMC'10, pages 86–90, 2010.

Appendix A

Extended Experimental Results

Table A.1 contains a reference for the program-interactions from example event sequences described in Chapter 8.

Table A.1: Program-Interaction Reference Table

program-interaction	Description
Comp-0	Build Java codebase
Comp-1	Clean compiled code
Copy-0	Copy 250mb file
Crypt-0	DES encrypt/decrypt text
Crypt-1	3DES encrypt/decrypt text
Crypt-2	AES256 encrypt/decrypt text
Crypt-3	RC4 encrypt/decrypt text
Crypt-4	DES encrypt/decrypt file
Crypt-5	3DES encrypt/decrypt file
Crypt-6	AES256 encrypt/decrypt file
Crypt-7	RC4 encrypt/decrypt file
Empty-0	Sleep 1s
Excel-0	Find
Excel-1	Replace
Excel-2	Sort
Excel-3	Add text Cell 1
Excel-4	Add picture Cell 1
Excel-5	Cell add link Cell 1
Excel-6	Cell remove link Cell 1
Excel-7 to Excel-10	Repeat previous four for Cell 2
Excel-11	Range 1 insert
Excel-12	Range 1 delete
Excel-13	Range 1 read
Excel-14	Range 1 copy
Excel-15	Range 1 format
Excel-16 to Excel-20	Repeat previous five for Range 2
FTP-0	FTP get file
FTP-1	FTP put file
IE-0	Navigate to URL
SQL-0	Execute SELECT query
VLC-0	Play 480p movie
Word-0	Export document
Word-1	Find text
Word-2	Replace text
Word-3	Add picture
Word-4	Save document as
Word-5	Get links in document
Word-6	Add link from doc start
Word-7	Write table from doc start
Word-8	Read table from doc start
Word-9	Insert text after from doc start
Word-10	Insert text before from doc start
Word-11	Format from doc start
Word-12	Compute statistics from doc start
Word-13	Delete text from doc start
Word-14 to Word-21	Repeat previous eight for doc end
Zip-0	Zip 50mb file
Zip-1	Unzip 50mb file

A.1 Evaluating the Exploratory Search

This section contains figures and tables related to the experiments examining the progress of the search methodology during the genetic algorithm evolutionary search for event sequences in Section 8.1.

Figure A.1 contains the configuration file for an experiment.

```
<Config>
  <Name>Progress_Hypothesis</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>AdAware</With>
    <Parameters/>
  </FitnessFunction>
  <Experiment>
    <PopulationSize>25</PopulationSize>
    <Generations>50</Generations>
    <MetricWeightManager>
      ...
    </MetricWeightManager>
    <SelectionProcess>Ranked_Based_Selection</SelectionProcess>
    <Operators>
      <Operator>
        <Name>Single_Point_Mutation</Name> <Weight>25</Weight>
      </Operator>
      <Operator>
        <Name>Two_Point_Crossover</Name> <Weight>75</Weight>
      </Operator>
    </Operators>
  </Experiment>
  <Progams>
    ...
  </Progams>
</Config>
```

Figure A.1: Experiment for Search Process Progress

A.1.0.1 Full Metric Suite Experiment

This subsection contains figures and tables from the search progress experiment for the full suite of metrics experiment in Section 8.1. Figures A.2 to A.7 show kernel density estimate plots, which are smoothed versions of histograms. It should be noted that there is no sample for the allocated pages metric, since for this experiment the allocated pages did not vary and thus were canceled out of any fitness function calculation.

Figure A.2 shows the script time distribution for the full suite experiment. This script time distribution is naturally very similar to that seen previously in Figure 8.3 for the single metric experiment, since both are a sample of the same metric just for two different experiments.

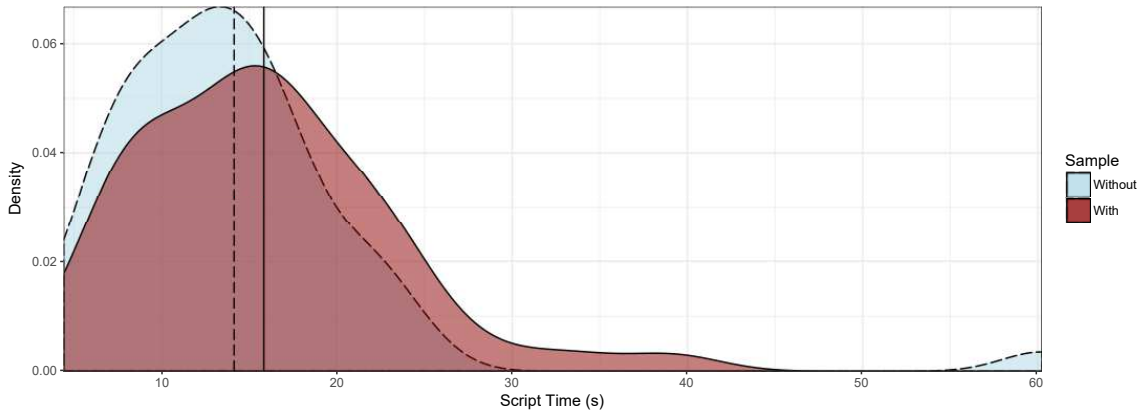


Figure A.2: Multi-Metric Experiment: Script Time Distribution Density

The metric of script execution time fills only one of the five categories of metrics for the full suite of metrics experiment. The other four of the five categories include: hard drive (HDD), network, RAM, and CPU metrics. It can be observed that a number of these metrics do not feature normal distributions. The reasoning for the continued use of these metrics was addressed in Section 5.3.

Figure A.3 contains the density plots for the category of Hard Drive (HDD) metrics. These include read/written bytes, Direct Memory Access(DMA) transfers, and Programmed Input/Output(PIO) transfers.

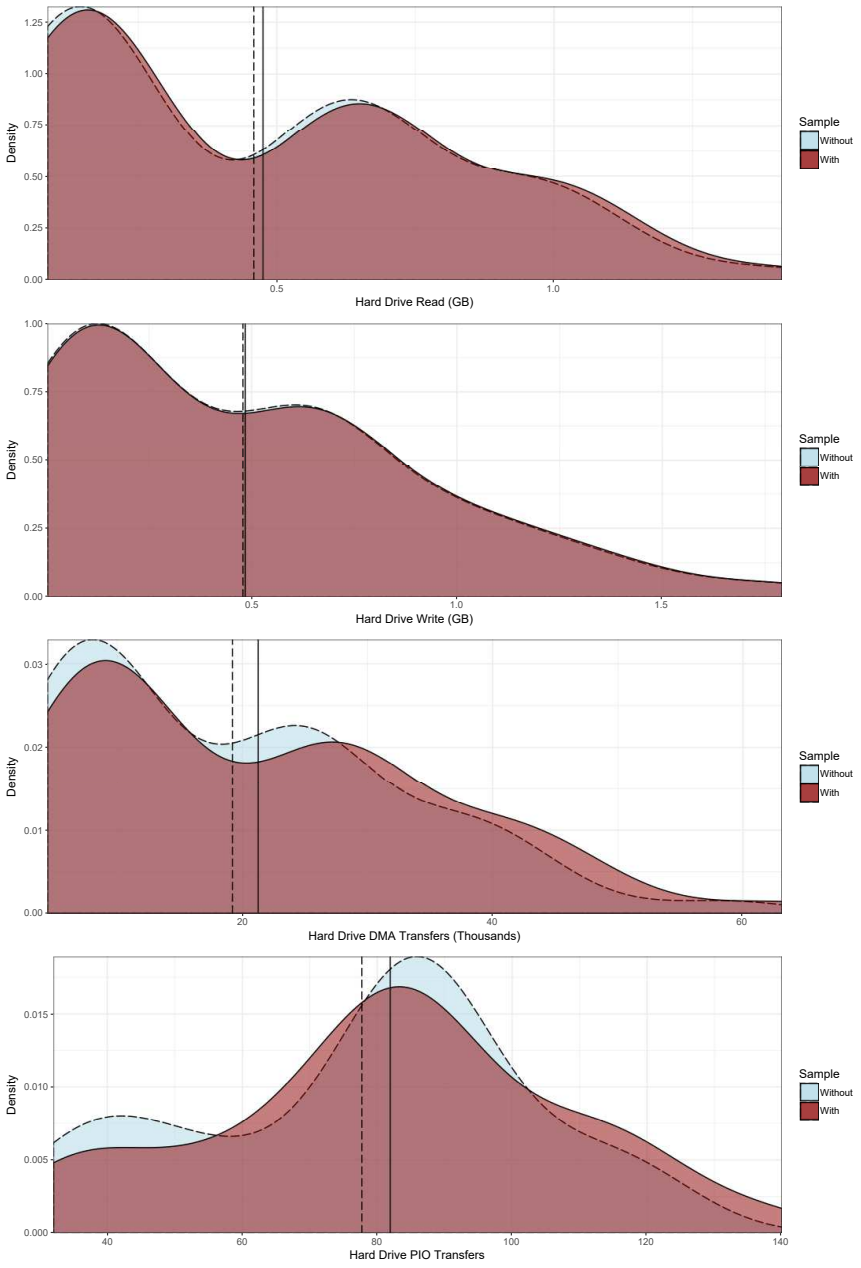


Figure A.3: Multi-Metric Experiment: HDD Distribution Density

Figure A.4 contains the density plots for the category of Network metrics. These include sent/received bytes and sent/received rates.

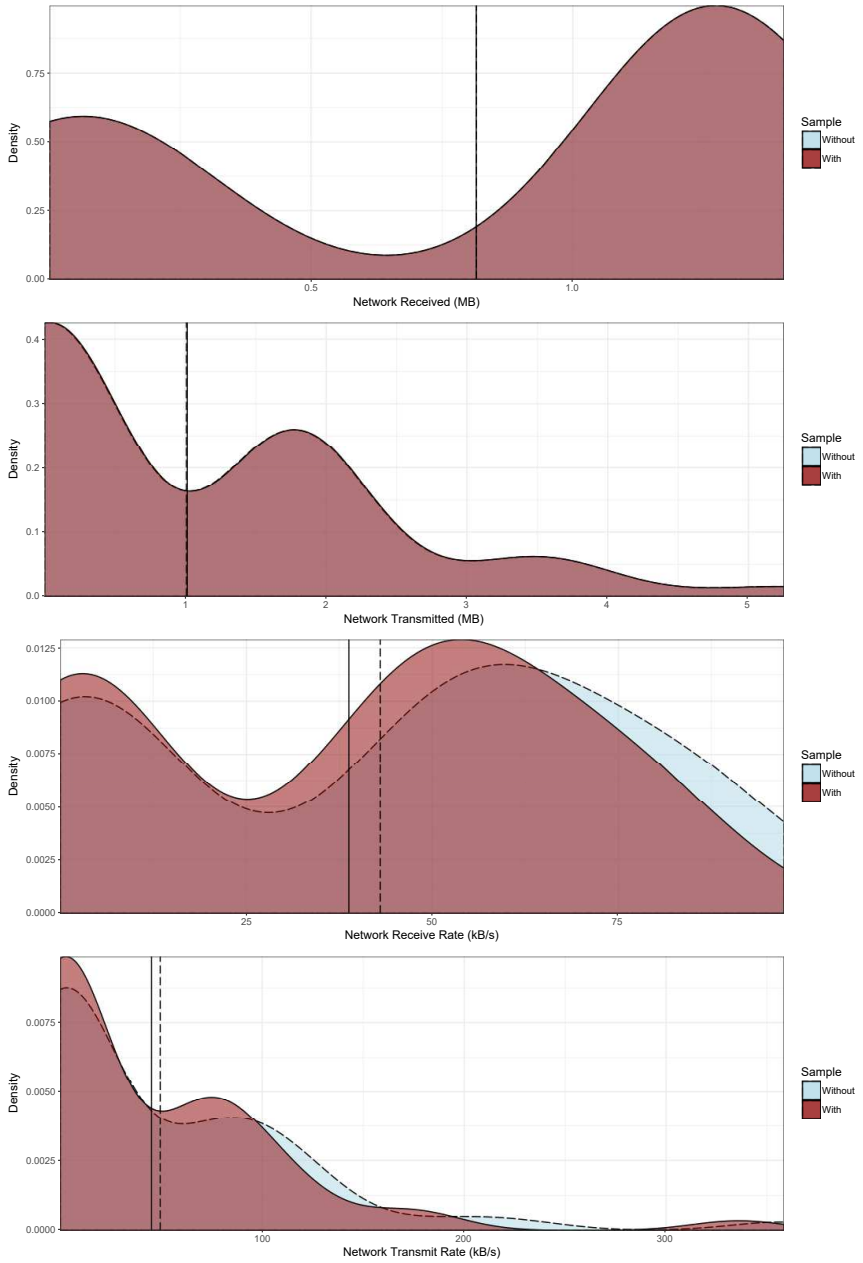


Figure A.4: Multi-Metric Experiment: Network Distribution Density

Figure A.5 contains the density plots for the category of RAM metrics. These include the average and minimum bytes of cache/free RAM.

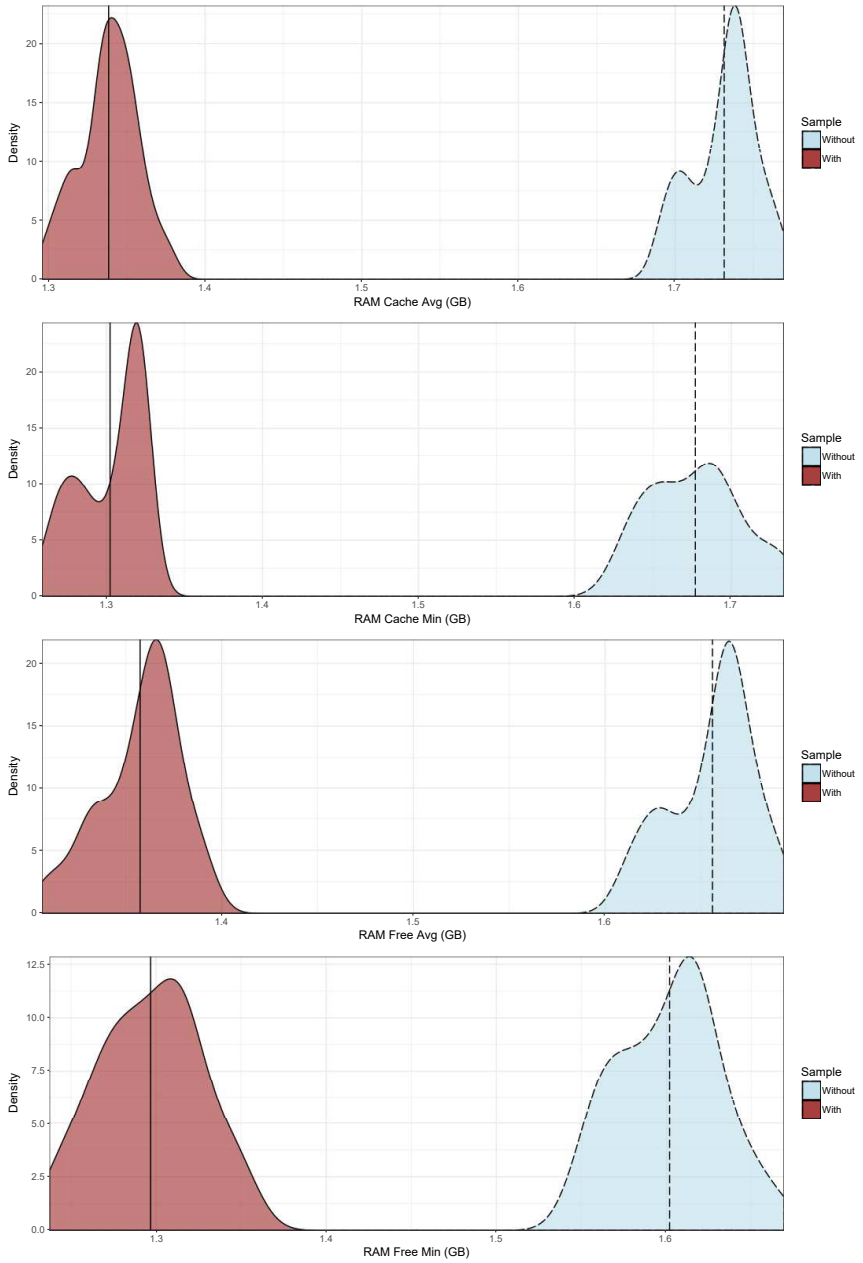


Figure A.5: Multi-Metric Experiment: RAM Distribution Density

Figures A.7 and A.6 contains the density plots for the category of CPU metrics. These include the average and maximum percent of CPU time spent in kernel/user operation and percent of time halted.

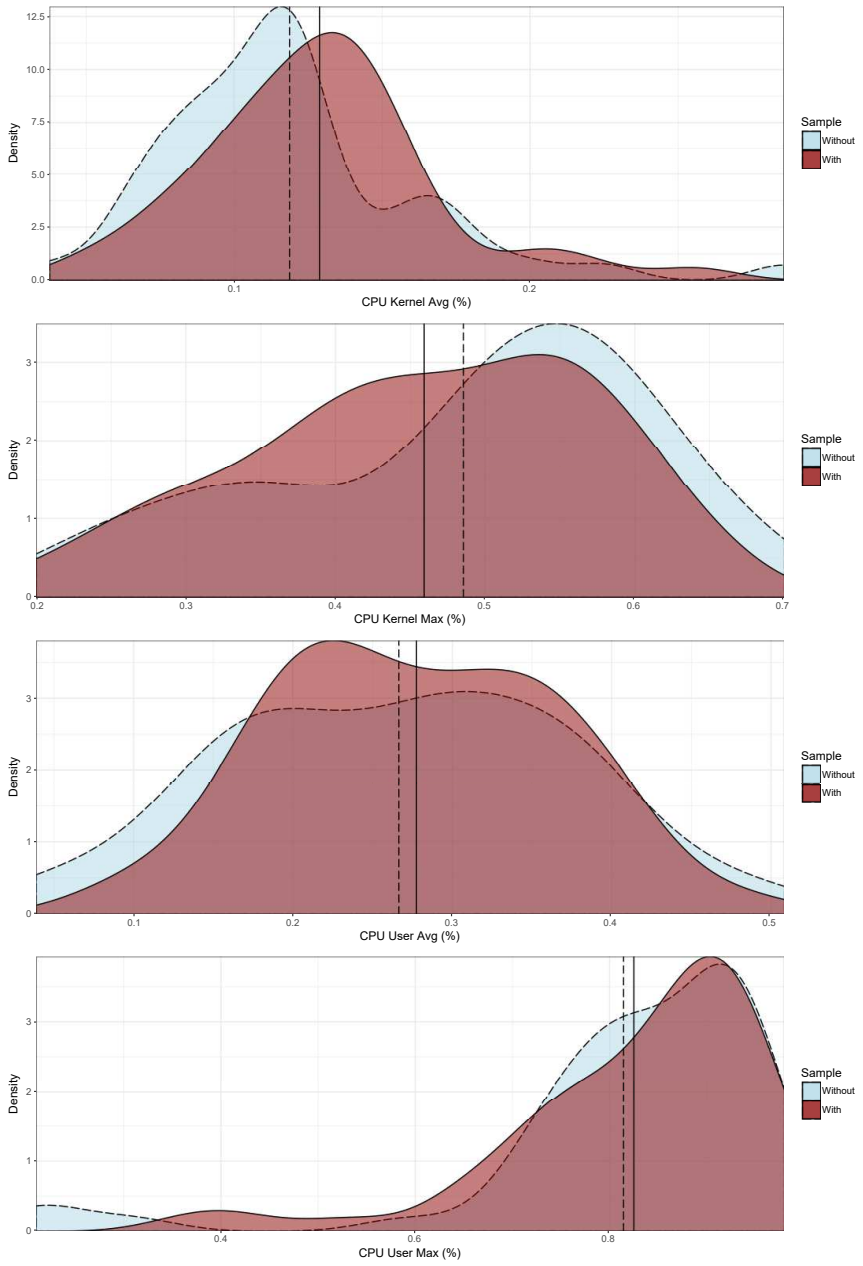


Figure A.6: Multi-Metric Experiment: CPU Distribution Density

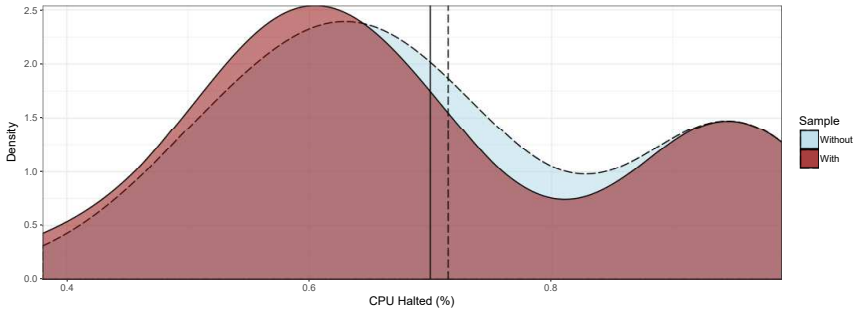


Figure A.7: Multi-Metric Experiment: CPU Halted Distribution Density

Figure A.8 shows the ID numbers in the population for every generation for the full suite of metrics experiment in Section 8.1.

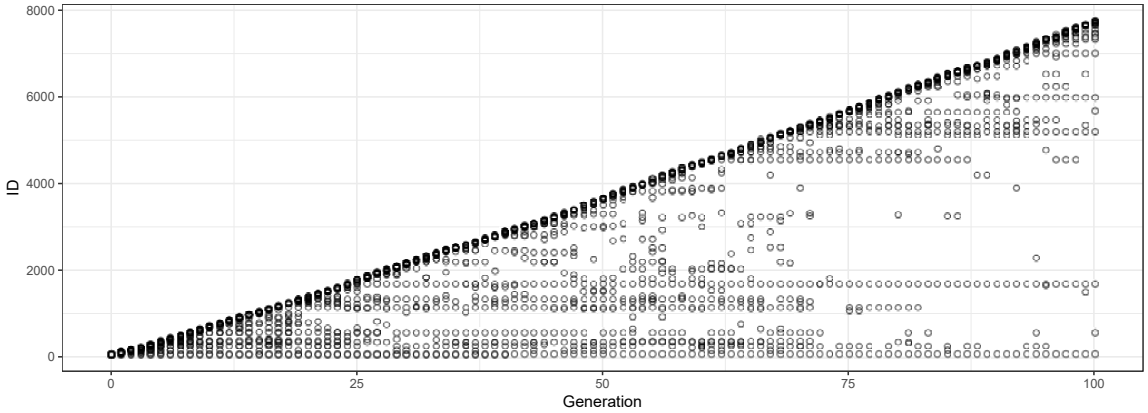


Figure A.8: Multi-Metric Experiment: Individual ID

A.2 Selecting a Single Security Mechanism

This section contains figures and tables related to the experiments examining the process to select between single security mechanisms completed in Section 8.2.

Figure A.9 contains the configuration file for the experiments.

```
<Config>
  <Name>Progress_Hypothesis</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>AdAware/Avast/ClamWin</With>
    <Parameters/>
  </FitnessFunction>
  <Experiment>
    <PopulationSize>25</PopulationSize>
    <Generations>100</Generations>
    <MetricWeightManager>
      <Weight name="script_time">1.0</Weight>
    </MetricWeightManager>
    <SelectionProcess>Ranked_Based_Selection</SelectionProcess>
    <Operators>
      <Operator>
        <Name>Single_Point_Mutation</Name> <Weight>25</Weight>
      </Operator>
      <Operator>
        <Name>Two_Point_Crossover</Name> <Weight>75</Weight>
      </Operator>
    </Operators>
  </Experiment>
  <Progams>
    ...
  </Progams>
</Config>
```

Figure A.9: Experiment for Single Security Mechanism

Figure A.10 contains the configuration file for the tests to examine the event sequence examples.

```
<Config>
  <Name>Single_Security_Mechanism_Test</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>Security_Mechanism_Instance</With>
    <Parameters/>
  </FitnessFunction>
  <Test>
    <Script>Event_Sequence</Script>
    <Count>25</Count>
  </Test>
  <Progams>
    ...
  </Progams>
</Config>
```

Figure A.10: Event Sequence Test for Single Mechanism Process

A.2.1 Experiment Results

Table A.2 holds the raw distribution values for Usage Profile 1.

Table A.2: Usage Profile 1: Sample m^{time} Statistics in s

stat	*Without*	*ClamWin*	*AdAware*	*Avast*
median	19.31	18.46	28.17	23.28
mean	18.84	18.38	28.72	23.19
st.dev.	4.87	4.55	8.54	4.99
max	33.35	31.13	57.87	38.91
min	8.43	8.47	10.43	13.07

Table A.3 and Figure A.11 show the raw distribution values from Table A.2 as values relative to the **Without** virtual machine values.

Table A.3: Usage Profile 1: Sample m^{time} Relative Statistics

stat	*ClamWin*	*AdAware*	*Avast*
median	98.0%	149.5%	123.6%
mean	97.5%	152.4%	123.1%
st.dev.	24.2%	45.3%	26.5%
max	165.2%	307.1%	206.5%
min	44.9%	55.4%	69.3%

Table A.4 shows the raw example event sequence script execution time values for Usage Profile 1 for the experiment in Section 8.2.

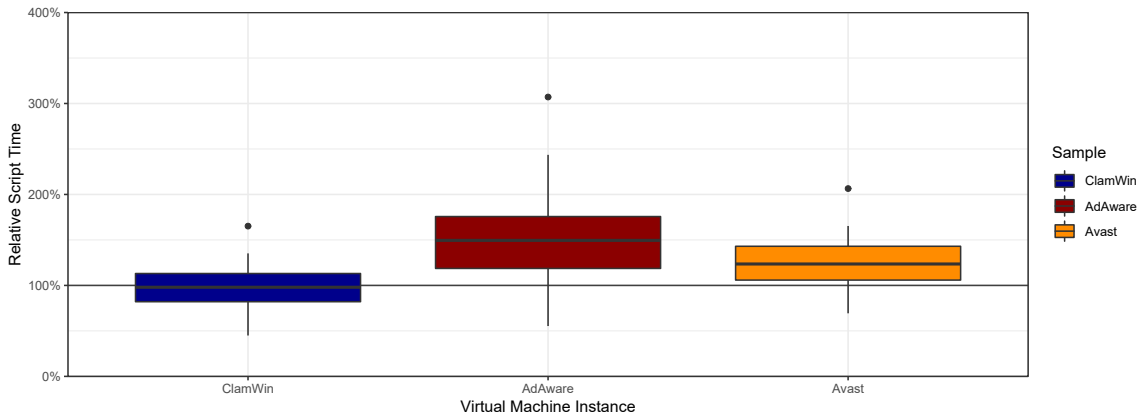


Figure A.11: Usage Profile 1: Sample m^{time} Relative Statistics

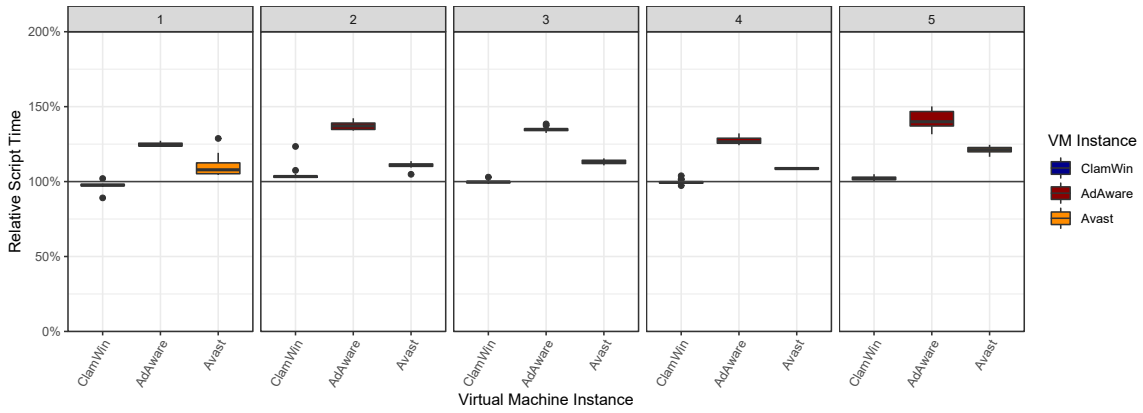
Table A.4: Usage Profile 1: Event Sequence Mean (St.Dev.) for m^{time} in s

event_seq	*Without*	*ClamWin*	*AdAware*	*Avast*
$clamwin_1$	19.6 (1.12)	19.1 (0.42)	24.4 (0.25)	21.5 (1.21)
$clamwin_2$	16.6 (0.13)	17.3 (0.71)	22.7 (0.42)	18.4 (0.29)
$clamwin_3$	17.9 (0.19)	17.9 (0.17)	24.1 (0.24)	20.3 (0.25)
$clamwin_4$	19.0 (0.21)	19.0 (0.25)	24.2 (0.44)	20.7 (0.12)
$clamwin_5$	14.5 (0.22)	14.8 (0.17)	20.6 (0.82)	17.6 (0.35)
$adaware_1$	17.7 (1.15)	16.1 (0.10)	21.0 (0.59)	19.1 (0.45)
$adaware_2$	15.0 (0.09)	14.7 (0.11)	18.6 (0.13)	17.5 (0.07)
$adaware_3$	16.9 (0.03)	16.8 (0.05)	20.3 (0.06)	19.5 (0.76)
$adaware_4$	16.5 (0.07)	16.3 (0.13)	20.6 (0.22)	18.6 (0.11)
$adaware_5$	16.4 (0.04)	16.4 (0.07)	20.3 (0.21)	20.2 (2.06)
$avast_1$	17.0 (0.19)	16.9 (0.08)	22.0 (0.73)	20.0 (0.12)
$avast_2$	14.1 (0.07)	14.1 (0.30)	19.3 (0.10)	18.0 (1.54)
$avast_3$	16.1 (0.06)	16.0 (0.05)	20.8 (0.14)	19.4 (0.67)
$avast_4$	13.4 (0.06)	13.4 (0.10)	18.3 (0.10)	16.8 (1.06)
$avast_5$	16.1 (0.30)	16.0 (0.18)	21.2 (0.70)	19.5 (0.28)

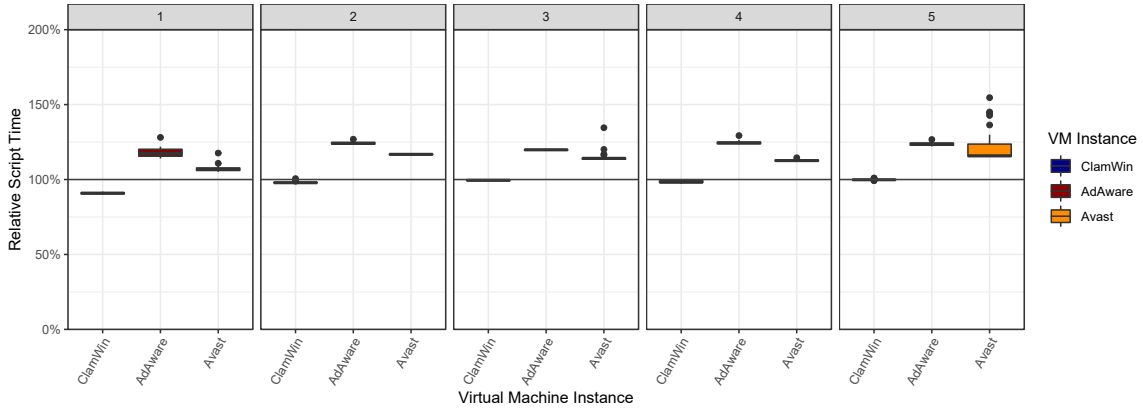
Table A.5 and Figure A.12 show the raw security mechanism values from Table A.4 as values relative to the **Without** virtual machine values.

Table A.5: Usage Profile 1: Event Sequence m^{time} Relative to Without

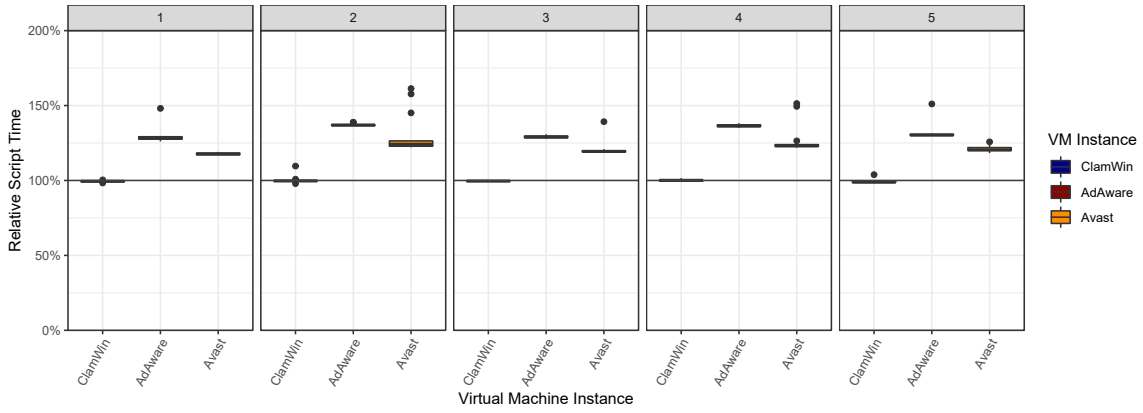
event_seq	*ClamWin*	*AdAware*	*Avast*
$clamwin_1$	97.4%	124.7 %	109.9 %
$clamwin_2$	104.4%	137.3 %	110.8 %
$clamwin_3$	99.8%	134.8 %	113.1 %
$clamwin_4$	99.7%	127.4 %	108.7 %
$clamwin_5$	102.1%	141.5 %	120.9 %
$adaware_1$	90.9%	118.1 %	107.5 %
$adaware_2$	98.1%	124.3 %	116.8 %
$adaware_3$	99.5%	119.8 %	115.3 %
$adaware_4$	98.6%	124.8 %	112.7 %
$adaware_5$	99.9%	123.8 %	122.7 %
$avast_1$	99.4%	129.1 %	117.8 %
$avast_2$	100.1%	137.1 %	128.1 %
$avast_3$	99.6%	129.1 %	120.3 %
$avast_4$	100.1%	136.5 %	125.6 %
$avast_5$	99.2%	131.3 %	121.0 %



(a) ClamWin Experiment



(b) AdAware Experiment



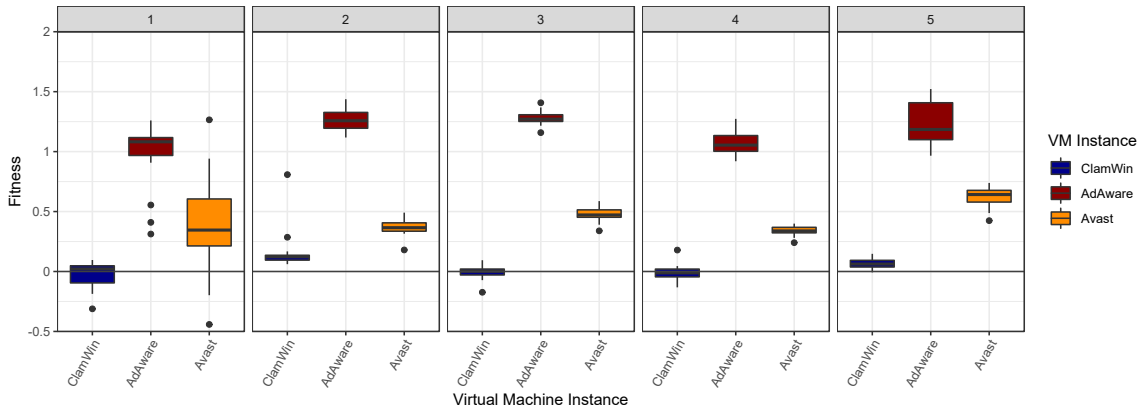
(c) Avast Experiment

Figure A.12: Usage Profile 1: Event Sequence m^{time} Relative to Without

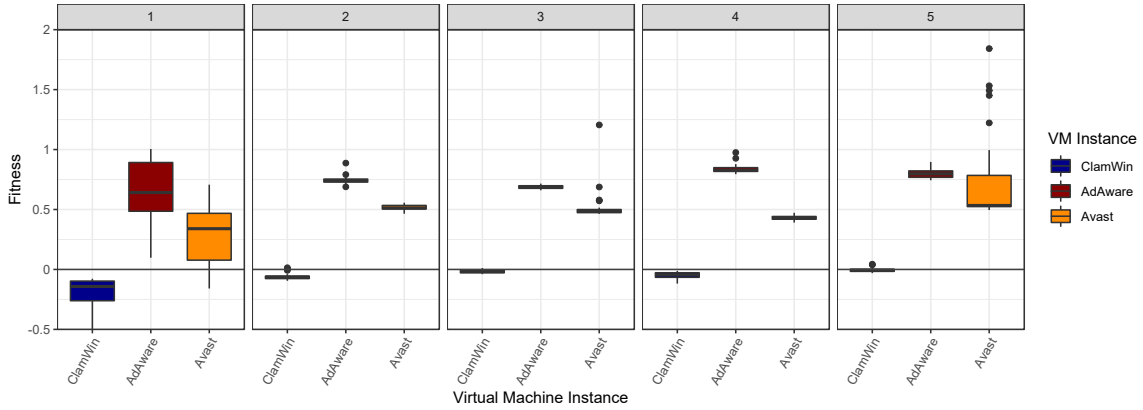
Table A.6 and Figure A.13 show the raw security mechanism values from Table A.4 as fitness values relative to the distribution values given in Table A.2.

Table A.6: Usage Profile 1: Event Sequence m^{time} Fitness

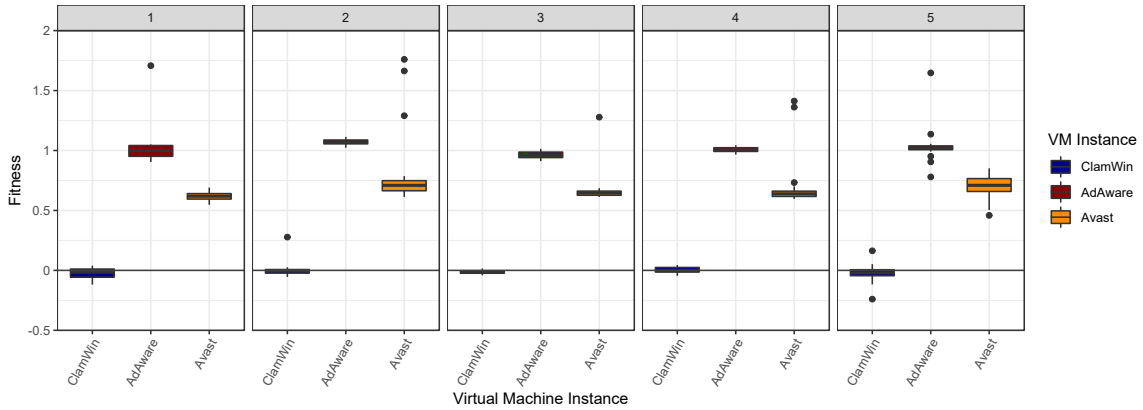
$event\_seq$	$ClamWin$	$AdAware$	$Avast$
$clamwin_1$	−0.10	0.99	0.40
$clamwin_2$	0.15	1.27	0.37
$clamwin_3$	−0.01	1.28	0.48
$clamwin_4$	−0.01	1.07	0.34
$clamwin_5$	0.06	1.24	0.62
$adaware_1$	−0.33	0.66	0.27
$adaware_2$	−0.06	0.75	0.52
$adaware_3$	−0.02	0.69	0.53
$adaware_4$	−0.05	0.84	0.43
$adaware_5$	0.00	0.80	0.76
$avast_1$	−0.02	1.02	0.62
$avast_2$	0.00	1.07	0.81
$avast_3$	−0.01	0.96	0.67
$avast_4$	0.00	1.01	0.70
$avast_5$	−0.03	1.04	0.70



(a) ClamWin Experiment



(b) AdAware Experiment



(c) Avast Experiment

Figure A.13: Usage Profile 1: Event Sequence m^{time} Fitness

Table A.7 holds the raw distribution values for Usage Profile 2 for the experiment in Section 8.2.

Table A.7: Usage Profile 2: Sample m^{time} Statistics in s

stat	*Without*	*ClamWin*	*AdAware*	*Avast*
median	32.96	32.91	34.58	38.19
mean	33.12	33.03	35.78	38.52
st.dev.	12.65	12.78	13.39	13.75
max	74.74	73.36	74.21	78.45
min	6.09	6.07	6.59	9.10

Table A.8 and Figure A.14 show the raw distribution values from Table A.7 as values relative to the **Without** virtual machine values.

Table A.8: Usage Profile 2: Sample m^{time} Relative Statistics

stat	*ClamWin*	*AdAware*	*Avast*
median	99.3%	104.4%	115.3%
mean	99.7%	108.0%	116.3%
st.dev.	38.6%	40.4%	41.5%
max	221.5%	224.0%	236.8%
min	18.3%	19.9%	27.5%

Table A.9 shows the raw example event sequence script execution time values for Usage Profile 2 for the experiment in Section 8.2.

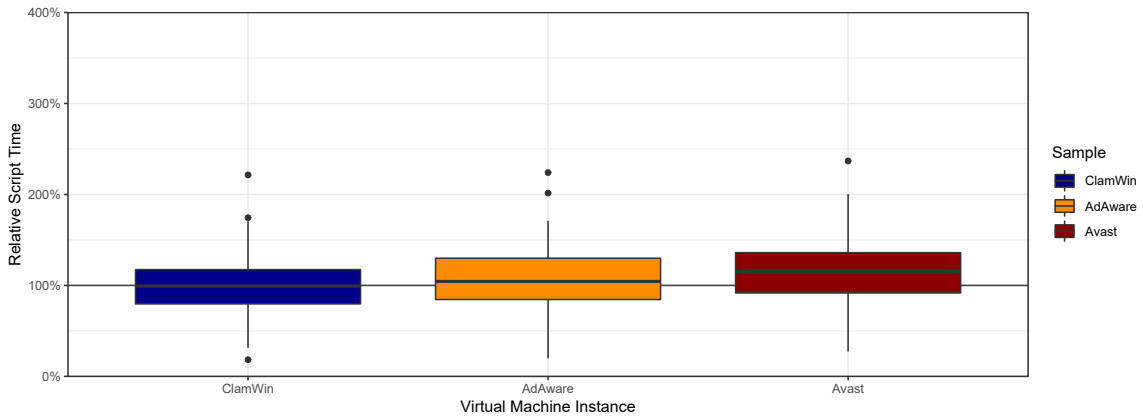


Figure A.14: Usage Profile 2: Sample m^{time} Relative Statistics

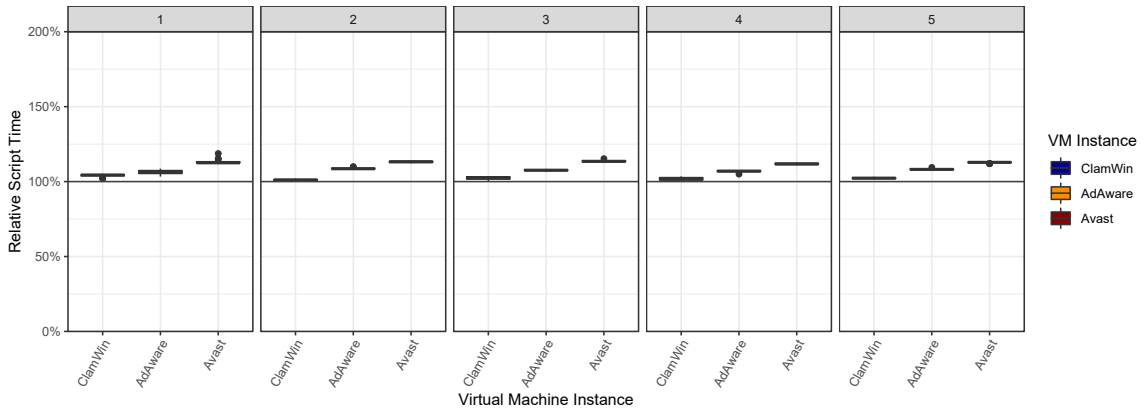
Table A.9: Usage Profile 2: Event Sequence Mean (St.Dev.) for m^{time} in s

event_seq	*Without*	*ClamWin*	*AdAware*	*Avast*
$clamwin_1$	45.7 (0.37)	47.6 (0.40)	48.6 (0.62)	51.7 (0.70)
$clamwin_2$	66.6 (0.25)	67.2 (0.40)	72.4 (0.34)	75.4 (0.34)
$clamwin_3$	55.5 (0.41)	56.8 (0.52)	59.6 (0.26)	63.0 (0.37)
$clamwin_4$	50.4 (0.35)	51.3 (0.48)	53.9 (0.31)	56.4 (0.28)
$clamwin_5$	62.3 (0.29)	63.7 (0.28)	67.4 (0.33)	70.2 (0.25)
$adaware_1$	28.2 (0.21)	28.0 (0.26)	30.6 (0.23)	33.4 (0.22)
$adaware_2$	36.7 (0.28)	36.7 (0.28)	40.3 (0.45)	43.2 (0.25)
$adaware_3$	37.0 (1.53)	36.7 (0.20)	40.2 (0.26)	42.9 (0.32)
$adaware_4$	36.5 (0.20)	36.5 (0.19)	40.2 (0.26)	43.1 (0.32)
$adaware_5$	30.0 (0.12)	30.1 (0.16)	33.0 (0.17)	35.8 (0.31)
$avast_1$	3.5 (0.03)	3.5 (0.03)	3.8 (0.08)	5.2 (0.05)
$avast_2$	12.1 (0.08)	12.1 (0.05)	13.2 (0.11)	15.4 (0.10)
$avast_3$	7.3 (0.18)	7.2 (0.36)	7.1 (0.23)	8.5 (0.11)
$avast_4$	3.5 (0.13)	3.4 (0.12)	3.8 (0.03)	5.1 (0.04)
$avast_5$	7.4 (0.24)	7.2 (0.32)	7.1 (0.14)	8.4 (0.09)

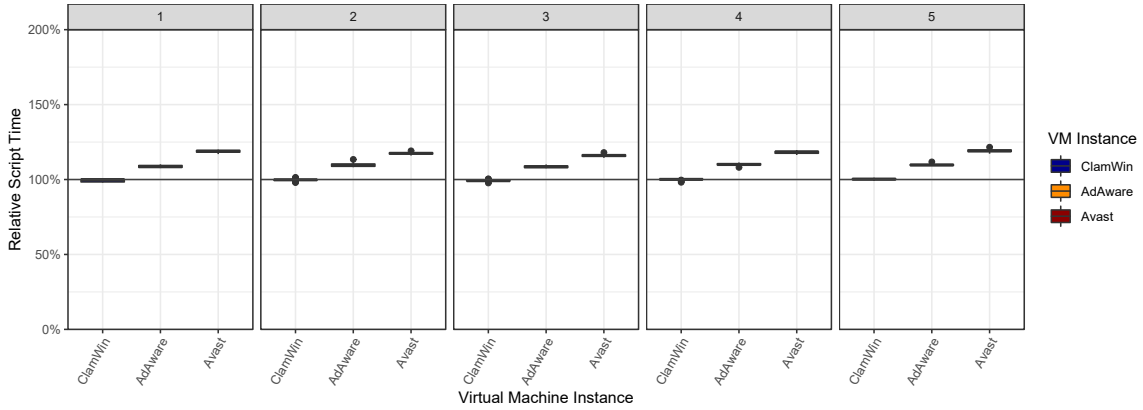
Table A.10 and Figure A.15 show the raw security mechanism values from Table A.9 as values relative to the **Without** virtual machine values.

Table A.10: Usage Profile 2: Event Sequence m^{time} Relative to Without

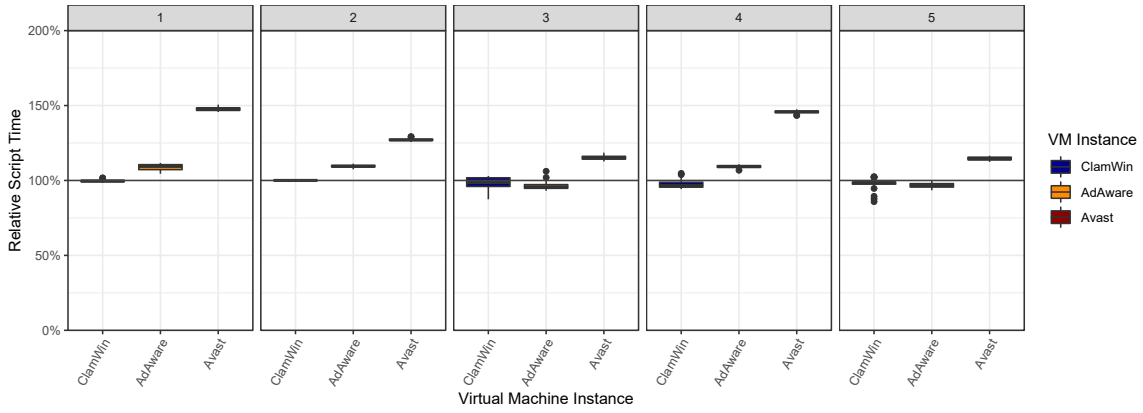
event_seq	*ClamWin*	*AdAware*	*Avast*
$clamwin_1$	104.1%	106.3 %	113.1 %
$clamwin_2$	101.0%	108.7 %	113.2 %
$clamwin_3$	102.4%	107.5 %	113.6 %
$clamwin_4$	101.9%	106.9 %	111.9 %
$clamwin_5$	102.3%	108.2 %	112.8 %
$adaware_1$	99.4%	108.8 %	118.8 %
$adaware_2$	99.8%	109.8 %	117.5 %
$adaware_3$	99.3%	108.6 %	116.1 %
$adaware_4$	100.0%	110.1 %	118.2 %
$adaware_5$	100.2%	109.8 %	119.2 %
$avast_1$	99.8%	108.9 %	147.7 %
$avast_2$	100.1%	109.4 %	127.2 %
$avast_3$	97.7%	96.6 %	115.3 %
$avast_4$	97.9%	109.3 %	145.7 %
$avast_5$	97.5%	96.7 %	114.5 %



(a) ClamWin Experiment



(b) AdAware Experiment



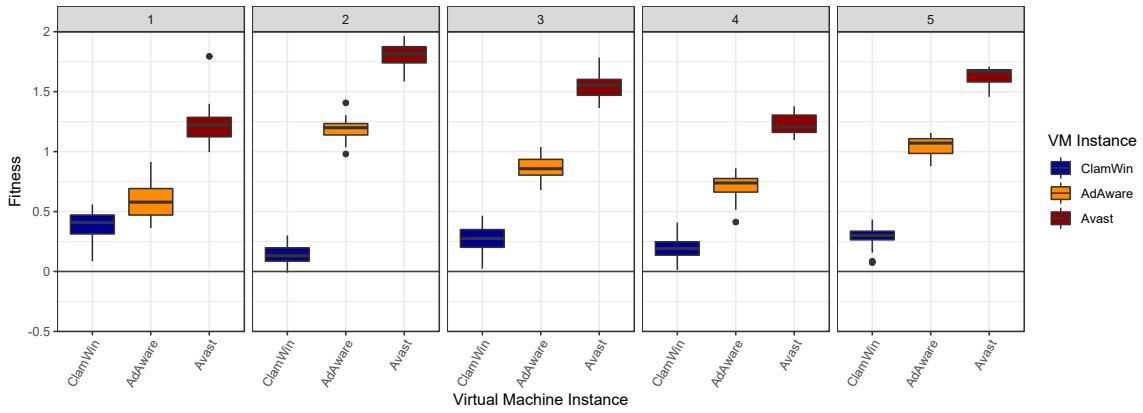
(c) Avast Experiment

Figure A.15: Usage Profile 2: Event Sequence m^{time} Relative to Without

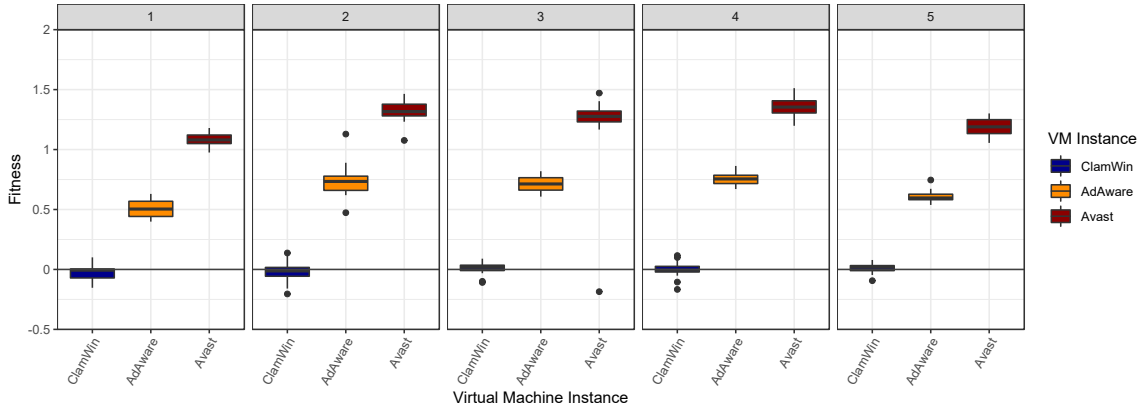
Table A.11 and Figure A.16 show the raw security mechanism values from Table A.9 as fitness values relative to the distribution values given in Table A.7.

Table A.11: Usage Profile 2: Event Sequence m^{time} Fitness

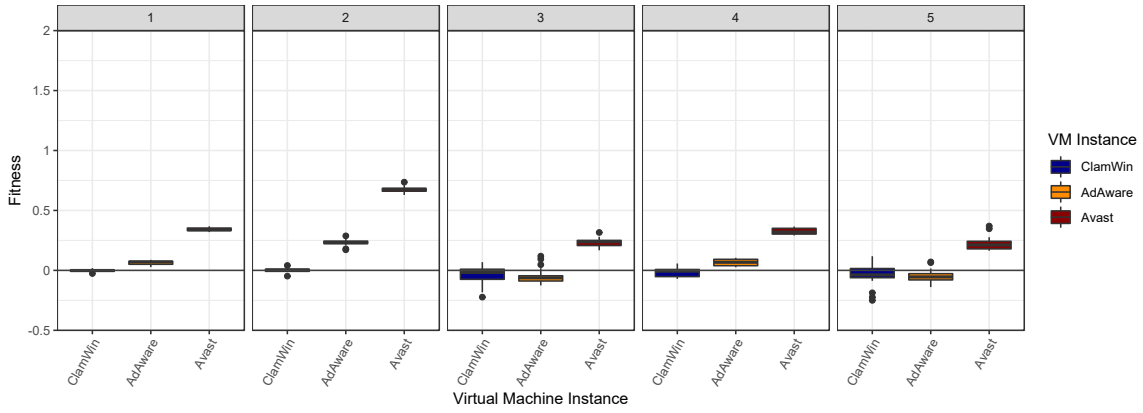
event_seq	*ClamWin*	*AdAware*	*Avast*
$clamwin_1$	0.39	0.59	1.23
$clamwin_2$	0.14	1.19	1.80
$clamwin_3$	0.27	0.86	1.55
$clamwin_4$	0.20	0.71	1.23
$clamwin_5$	0.29	1.05	1.63
$adaware_1$	−0.03	0.51	1.08
$adaware_2$	−0.02	0.74	1.32
$adaware_3$	−0.05	0.65	1.22
$adaware_4$	0.00	0.76	1.36
$adaware_5$	0.01	0.60	1.19
$avast_1$	0.00	0.06	0.34
$avast_2$	0.00	0.23	0.67
$avast_3$	−0.03	−0.05	0.23
$avast_4$	−0.01	0.07	0.33
$avast_5$	−0.04	−0.05	0.22



(a) ClamWin Experiment



(b) AdAware Experiment



(c) Avast Experiment

Figure A.16: Usage Profile 2: Event Sequence m^{time} Fitness

A.3 Examining Subsets of Security Mechanisms

This section contains figures and tables related to the experiments examining the process to select between subsets of security mechanisms completed in Section 8.3.

Figure A.17 contains the configuration file for the experiments.

```
<Config>
  <Name>Subset_Security_Mechanisms_Experiment</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>Subset_Mechanisms_Instance</With>
    <Parameters/>
  </FitnessFunction>
  <Experiment>
    <PopulationSize>25</PopulationSize>
    <Generations>50</Generations>
    <MetricWeightManager>
      ...
    </MetricWeightManager>
    <SelectionProcess>Ranked_Based_Selection</SelectionProcess>
    <Operators>
      <Operator>
        <Name>Single_Point_Mutation</Name> <Weight>25</Weight>
      </Operator>
      <Operator>
        <Name>Two_Point_Crossover</Name> <Weight>75</Weight>
      </Operator>
    </Operators>
  </Experiment>
  <Progams>
    ...
  </Progams>
</Config>
```

Figure A.17: Experiment for Subset of Security Mechanisms

Figure A.18 contains the configuration file for the tests to examine the event sequence examples.

```
<Config>
  <Name>Subset_Security_Mechanisms_Test</Name>
  <Seed>12345</Seed>
  <Dimensions>10</Dimensions>
  <SampleSize>50</SampleSize>
  <Top>100</Top>
  <FitnessFunction>
    <Name>Fitness_Function</Name>
    <VM>Virtual_Machine</VM>
    <Without>Fresh_Instance</Without>
    <With>Subset_Mechanisms_Instance</With>
    <Parameters/>
  </FitnessFunction>
  <Test>
    <Script>Event_Sequence</Script>
    <Count>25</Count>
  </Test>
  <Progams>
    ...
  </Progams>
</Config>
```

Figure A.18: Event Sequence Test for Multiple Mechanisms Process

A.3.1 Experiment Results

Table A.12 holds the raw script execution time distribution values for the experiment.

Table A.12: Subsets: Sample m^{time} Statistics in m

stat	*Without*	*Four*	*Five*	*Six*
median	0.42	0.81	0.96	1.00
mean	0.43	0.87	0.96	1.23
st.dev.	0.17	0.37	0.41	0.66
max	1.50	2.13	2.53	4.13
min	0.14	0.30	0.34	0.33

Table A.13 and Figure A.19 shows the raw script execution time distribution values from Table A.12 as values relative to the **Without** virtual machine values.

Table A.13: Subsets: Sample m^{time} Relative Statistics

stat	*Four*	*Five*	*Six*
median	189.7%	223.1%	233.7%
mean	201.8%	223.5%	286.5%
st.dev.	85.2%	95.6%	153.0%
max	71.1%	79.4%	77.6%
min	495.9%	589.3%	964.4%

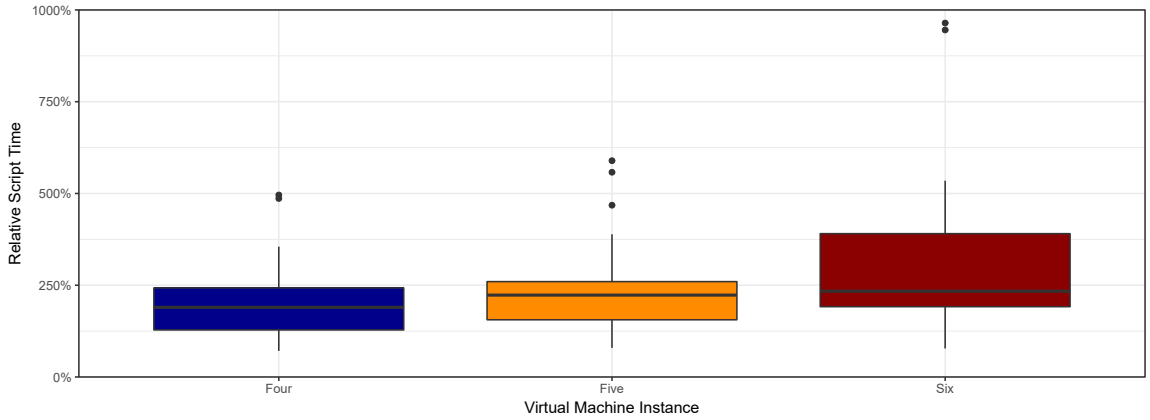


Figure A.19: Subsets: Sample m^{time} Relative Statistics

Table A.14 and holds the full metric suite of average distribution values for the experiment.

Table A.14: Subsets: Median Sample Statistics

CATEGORY	*HDD*				*NET*				*CPU*					*RAM*				*SCRIPT*
METRIC	*DMA*	*PIO*	*READ*	*WRITE*	*REC*	*TRANS*	*RX*	*TX*	*HALT*	*KERNAVG*	*KERNMAX*	*USERAVG*	*USERMAX*	*CACHAVG*	*CACHMIN*	*FREEAVG*	*FREEMIN*	*TIME*
	thous.		*GB*	*GB*	*MB*	*MB*	*Kb/s*	*Kb/s*	%	%	%	%	%	*GB*	*GB*	*GB*	*GB*	*m*
Without	19.49	101.21	0.47	0.49	0.77	0.90	25.75	29.59	51.2%	15.4%	54.0%	45.5%	96.1%	1.73	1.68	1.65	1.60	0.43
Four	26.99	158.14	0.65	0.49	0.83	1.01	14.82	19.00	21.7%	23.0%	75.8%	54.3%	98.0%	1.21	1.13	1.35	1.24	0.87
Five	29.13	169.35	0.70	0.49	0.85	0.95	13.88	16.03	27.3%	26.5%	77.8%	53.9%	97.7%	1.05	0.96	1.17	1.05	0.96
Six	39.04	184.29	0.82	0.51	0.83	1.03	12.25	15.59	10.0%	28.0%	76.8%	55.2%	97.5%	0.63	0.42	0.67	0.56	1.23

Figures A.20 to A.23 depict the distributions seen in Table A.14 as box plots instead of just averages.

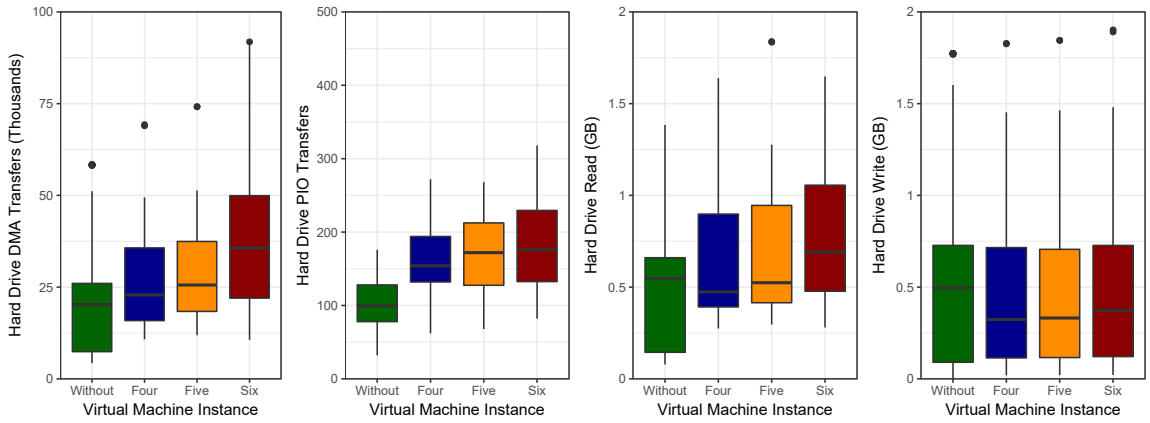


Figure A.20: Subsets: Sample HDD Statistics

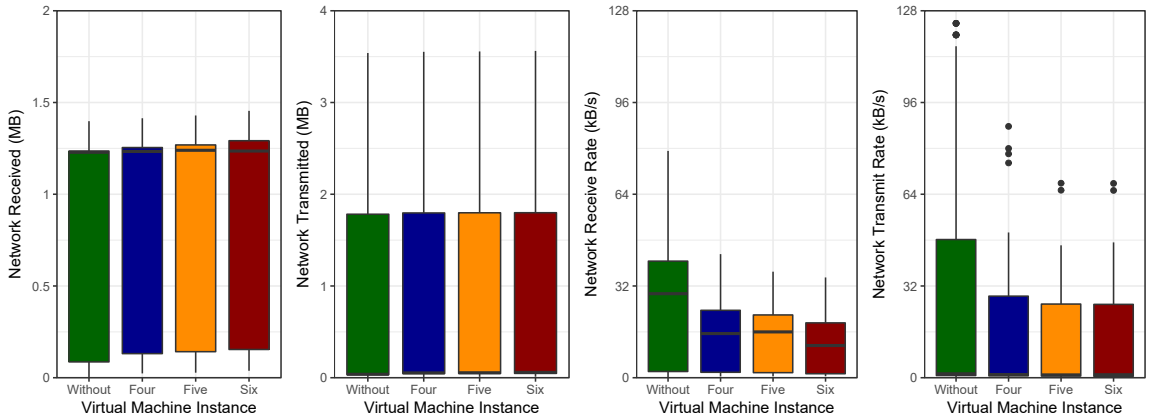


Figure A.21: Subsets: Sample Network Statistics

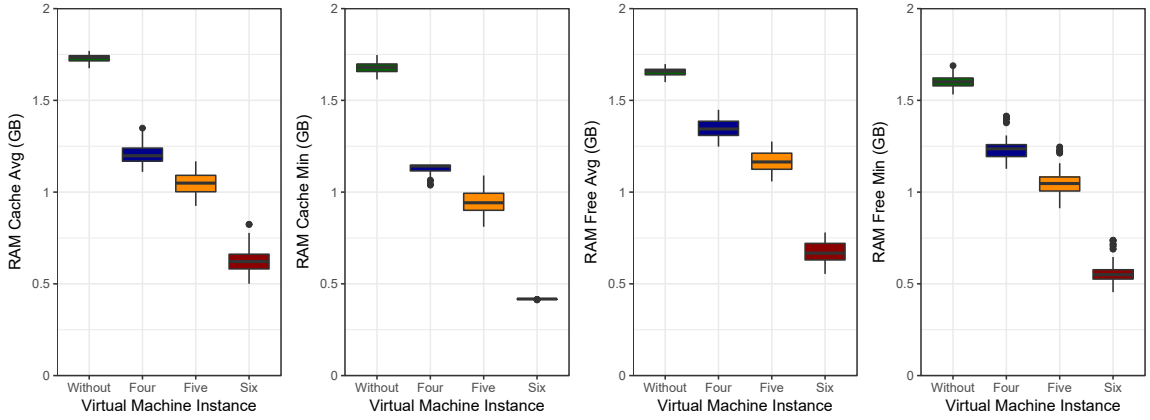


Figure A.22: Subsets: Sample RAM Statistics

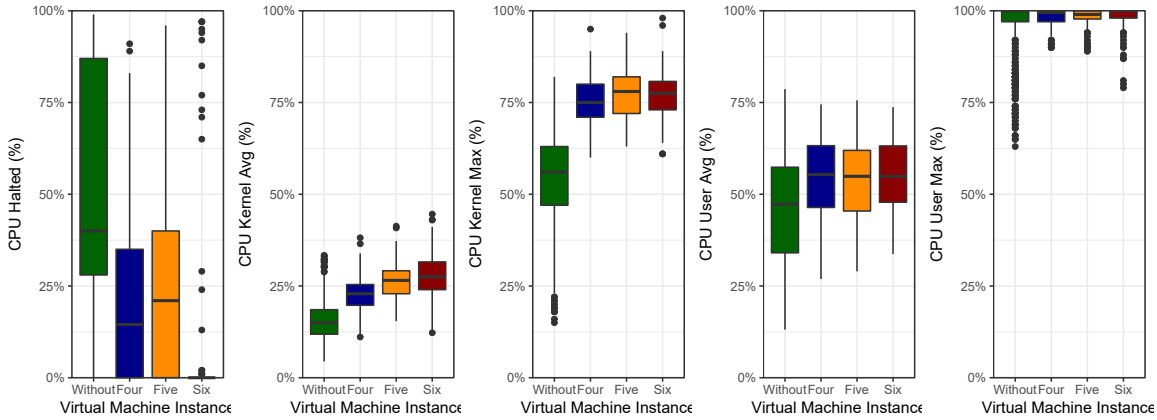


Figure A.23: Subsets: Sample CPU Statistics

Table A.15 shows the raw example event sequence script execution time values for the top example event sequence from each of the three previous experiments.

Table A.15: Subsets: Event Sequence Mean (St.Dev.) for m^{time} in s

event_seq	*Without*	*Four*	*Five*	*Six*
$four_1$	0.22 (0.010)	0.89 (0.097)	1.02 (0.214)	1.51 (0.233)
$five_1$	0.83 (0.008)	2.99 (0.123)	3.14 (0.043)	4.60 (0.162)
six_1	0.71 (0.007)	1.81 (0.028)	2.01 (0.029)	4.63 (0.666)

Table A.16 shows the raw security mechanism values from Table A.15 as values relative to the **Without** virtual machine values.

Table A.16: Susbets: Event Sequence m^{time} Relative to Without

event_seq	*Four*	*Five*	*Six*
$four_1$	402.7%	457 %	678.9 %
$five_1$	360.8%	377.7 %	554.7 %
six_1	256.1%	284.8 %	655.4 %

Table A.17 shows the comprehensive fitness values relative to the distribution values given in Table A.14.

Table A.17: Subsets: Event Sequence Comprehensive Fitness

event_seq	*Four*	*Five*	*Six*
$four_1$	3.19	3.20	3.38
$five_1$	4.22	3.68	3.90
six_1	1.59	1.61	5.53