Using Exploratory Testing for Decision Support in Choosing a Security Mechanism

Jonathan Hudson
Department of Computer Science
University of Calgary
Calgary, Canada
jwhudson@ucalgary.ca

Jörg Denzinger
Department of Computer Science
University of Calgary
Calgary, Canada
denzinge@ucalgary.ca

***Abstract*—From the point of view of a user, a security mechanism for a computer should protect it from the particular kind of attacks it is designed for, while influencing the performance of the computer for the user's applications as little as possible. In this paper, we present an evolutionary learning approach for exploratory testing of the performance consequences of a security mechanism on the user's usage profile, i.e. the applications the user is using.**

By learning application interaction sequences with performances that are the most negatively influenced by the installation of a security mechanism, a user can evaluate if the performance losses are acceptable and by applying our approach to several mechanisms with comparable protection, a user can make an informed decision which mechanism is better for him/her. As proof-of-concept, we used our approach to explore anti-virus security mechanisms operating in a Windows XP environment. Our experiments show that different usage profiles are indeed better served by different security mechanism.

***Index Terms*—testing, exploratory, security, performance**

I. Introduction

Over the last decades, the threat to information technology systems has steadily increased, creating an arms race between bad actors and the developers of security mechanisms and policies. As the increase in the number of serious data breaches and successful virus attacks shows, the bad actors have gained substantial ground. An ever-increasing problem regarding computer security is the need to integrate the technological aspects of security with the performance and functionality demands of the users of the system.

For any particular user the usefulness of a security mechanism depends on two criteria, namely the quality of the defence it offers against attacks and the possible performance challenges it poses for the user. Research has essentially exclusively focused on improving the quality of defence. However, even the best security mechanism (regardless of how "best" is defined) is useless, if a user deactivates it due to its disruptiveness in regard to the user's work requirements, i.e. the needed programs and the particular interactions with them. This avoidance of security mechanisms by individual users is part of the so-called shadow IT phenomenon (see [1]) and naturally can produce gaping holes in the overall defence strategy for a system.

In this paper, we present an exploratory search process using an evolutionary approach for learning interaction sequences with a set of programs. The goal of this search process is to identify sequences that are the most negatively influenced by installation of a particular security mechanism. This exploration is performed relative to the usage profile (programs and interactions with them) of a particular user. Seeing the produced interaction sequences for one security mechanism allows the user to decide if the performance losses due to the mechanism are acceptable. Cross-examination of event sequences produced for several mechanisms can allow a user to determine which of the mechanisms has the least losses for his/her usage needs.

While the evolutionary learner developed mainly uses the usual evolutionary operators on sequences, the key challenge for this problem is creating a fitness value for an interaction sequence. Following [2], [3], we evaluated an interaction sequence using simulations, in our case on a virtual machine, and therefore we had to deal with outlier behaviour originating from the host machine's interaction with the virtual machine. At the same time, we had to address general stochastic behaviour that would be found in both physical and virtual machine environments. Additionally, we are interested in the particular unique consequences of having a security mechanism installed and therefore the fitness measure compares a sequence execution with and without the mechanism. Finally, the method is designed to integrate several candidate performance measures.

We dealt with all these problems by initially creating two sample distributions for each performance measure using randomly generated interaction sequences. One distribution is created for the tested system without the security mechanism installed (the unsecured system), and the other with the mechanism (the secured system). The fitness function then evaluates an interaction sequence within the evolutionary search by comparing the performance measures with the corresponding distribution. For our fitness goal of valuing sequences higher that performed worse in the secured system versus the unsecured system, we then made use of these distribution relative measures. Additionally, when we re-evaluate sequences during subsequent generations of the search we combine each generational fitness value into an overall fitness by choosing the median value. For our experiments in this paper, we utilized run time as the chosen performance measure.

978-1-7281-2153-6/19/$31.00

Our proof-of-concept testing system targeted anti-virus software for the Windows XP operating system. We created two usage profiles by splitting our set of application programs in half and evaluated two active security mechanisms (and a passive one for additional comparison). Our testing system showed its ability to provide decision support to individual users by indicating a preference for a different security mechanism for each of the usage profiles.

This paper is organized as follows: after this introduction, in Section II we define the problem of effective security, followed by an abstract description of our evolutionary testing system in Section III. Section IV describes the design of our experiment with the testing system followed in Section V by the results of the experiment. Section VI presents the little related work there is and Section VII presents our conclusions and possible future work.

II. The Effective Security Problem

The Effective Security (ES) problem is the challenge of selecting a security mechanism that provides the desired functionality of the computer while minimizing the resultant cost to performance given that the mechanism provides the intended protection. The performance and functionality of the computing system will be evaluated according to metrics observed via interactions with a collection of programs. These metrics and possible interactions are representative of the performance and functionality loads indicated by a user's usage profile.

To define the ES problem, we begin by 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 $p_j = \{ip_{j,1}, ip_{j,2}, \ldots\}$. The set I^P contains all the possible interactions for the programs in P. The goal of the set P and associated set I^P is that they are representative of the usage profile of the user [4].

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 $es = (ev_1, ev_2, \ldots)$. 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 $ev = (ip_{j,k}, t)$. ES^P is the set of all possible interaction event sequences 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.

Performance and functionality are assessed according to a set $M = \{m_1, m_2, \ldots, m_o\}$ of o metrics. A metric $m \in M$ consists of a pair, such that $m = ([a, b], sgn)$ with $a, b \in \mathbb{R}$ and $sgn \in \{-1, +1, N/A\}$ where $[a, b]$ represents a range of performance or functionality which is acceptable. A value strict functional metric will have $sgn = N/A$, while a performance-functionality metric will have either $sgn = -1, +1$.

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

The simplest example of a performance-functionality metric is a measure of time for a sequence of interactions to complete. For example, we have a time metric m^{time} such that $m^{time} = ([0, b], -1)$. 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.

In the real world, an interaction event sequence is possibly infinite in length. It is also not possible to explore every one of the interaction event sequences found in ES^{eff} given the size of the set. The purpose of utilizing evolutionary search is to effectively discover and explore the interaction event sequences in ES^{eff} that are the most challenging to system performance.

Definition I. Effective Security (ES) Problem
The ES problem is the determination of an effective security mechanism S^{eff}. S^{eff} fulfills the desired performance functionality predicate $perf$ while minimizing the extent of unwanted performance loss in the effectiveness measure eff. The system's functionality and performance are examined under a set M of program metrics measured during the execution of user interaction event sequences from ES^{eff} representative of the user's performance requirements. □

The predicate $perf$ is used to decide if a security program achieves the desired absolute level of performance or functionality in regard to the performance metrics M for some interaction event sequence $es \in ES^{\mathit{eff}}$. This absolute value is a level of performance and functionality that, if violated, represents a system state which would not be acceptable as usable by the user. There is an assumption that this predicate should always be satisfied by the system state protected by no security mechanism S^0. In other words, S^0 should be such that $perf$ is true for all event sequences in ES^{eff}. Note, we will use $S^?$ to denote a prospective security mechanism we are examining.

Definition II. 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 ES^{eff}, then the goal is to minimize the extent of unwanted performance losses. That is, given a performance loss effectiveness measure eff, the goal is to avoid event sequences in ES^{eff} which maximize eff. An event sequence that creates performance load pressure on the secured system $S^?$ with the security mechanism installed may

equally create load on a completely unsecured system S^0. To differentiate, the effectiveness measure *eff* is a relative measure of the differential in performance between the secured system $S^?$ and the baseline unsecured system S^0.

Definition III. Effectiveness Measure (*eff*)
The effectiveness measure *eff* is the weighted summation of performance metrics M under the security mechanism in regards 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 which returns a differential 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 sequence es. □

The differential function *diff* returns an increasingly positive value representative of the performance loss between the measured and baseline values. The direction that the differential function *diff* considers performance loss is indicated by the sgn_m of each performance metric. Negative differentials returning less than 0 are ignored as 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 cancelling each other out upon summation. Purely functional metrics are not included in the effectiveness measure. The issue of performance being dynamic between different executions of the same interaction event sequence will be addressed in the definition of *diff* in Section III.

The performance predicate *perf* can be used by an evolutionary search to indicate event sequences where the functionality or performance of the secured system was unacceptable. At the same time, the effectiveness measure *eff* can be used as the fitness function to direct an evolutionary search. Together, a prospective security mechanism which passes the *perf* predicate and does not exhibit unwanted levels of performance loss for explored event sequences in ES^{Eff} may be considered as a viable candidate solution for the ES problem.

III. Evolutionary Search for Event Sequences

This section introduces the abstract formulation of an evolutionary search for interaction event sequences. This formulation utilizes the predicate *perf* to identify functionality failures in the examined system configuration and the effectiveness measure *eff* as a fitness function to direct the search. We will define the individual solutions, then explore the operators to manipulate solutions, and finally discuss the evolutionary fitness function. Particularly, we will elaborate on the definition of the differential function *diff* used in the effectiveness measure *eff*.

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

A. Individual

If we look at our definitions from the last section, then an obvious choice for an individual is a sequence of interaction events. But this representation faces a problem, namely that every event sequence needs to be examined by executing it in both systems $S^?$ and S^0. We have 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 systems. Even if two time values t_1 and t_2 had been chosen for two events such that $t_1 = t_2$, the delay between systems does not mean that they would be executed at the same time. Therefore we cannot use time values in our sequences representing individuals. Instead, 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 to execute a delay of a small period of time before proceeding to the next event.

Note that this does not mean that the consequences of triggering an event will not overlap. It just means that the initial triggers of events will not be simultaneous. This leads us to the following definition of an individual:

Definition IV. Individual (*indi*)
An individual *indi* for the ES problem is an integer sequence $indi = (j_1, k_1, \ldots, j_p, k_p)$ with $j_i \in \{1, \ldots, |P|\}$ and $k_i \in \{1, \ldots, |p_{j_i}|\}$ for $p_{j_i} \in P$. □

In order to allow for the already mentioned *nop*, we represent it by program 0 and interaction 0.

The initial evolutionary algorithm population consists of a count of fixed pop_size random solutions. Each random solution is formed from a random sequence of integer pairs. Each pair begins with the selection of a pseudo-random positive integer selected from a uniform distribution in the size of P to indicate a program. A second integer is then chosen from a uniform distribution in the size of that program's interaction set to indicate which specific action will form that pair.

B. Genetic Operators

To create new individuals for subsequent generations in order to explore the search space it is necessary to define the single-point mutation operator mut_{op} and two-point crossover operator $cross_{op}$ for the particular problem of Effective Security. These operators each use a rank-based selection process sel_{proc} to select an individual from the current population [5].

Definition V. Single-Point Mutation Operator (op_{mut})
The single-point mutation operator op_{mut} is a genetic operator $op_{mut}(indi_{parent}) = indi_{mutation}$ 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 count of indices. The interaction event $ip_{j,k} \in I^P$ at one 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 $indi_{mutation}$. □

Definition VI. Two-Point Crossover Operator (op_{cross}) The two-point crossover operator op_{cross} is a genetic operator $op_{cross}(indi^1_{parent}, indi^2_{parent}) = indi_{crossover}$ where two parent individuals $indi^1_{parent} = (j_1, k_1, \ldots, j_p, k_p)$ and $indi^2_{parent} = (j'_1, k'_1, \ldots, j'_p, k'_p)$ are selected using sel_{proc}. A sequence from the two indices m and n is filled with p'_i and k'_i from $indi^2_{parent}$ while the remainder is filled with p_i and k_i from $indi^1_{parent}$. The result is the child individual $indi_{crossover}$. □

These fundamental operators are used to create children for the next generation, however there are additional useful operators. One such additional operator op_{prev_best} maintains the previous generation's most fit solution. This operator is used once a generation and simply determines the solution with the highest fitness in the previous generation and promotes a copy of it into the following generation. This fitness is based on the effectiveness measure *eff*. There are two variants of this operator, one that propagates the individual with the highest maximum fitness which will result in re-evaluation of outliers, and one for the highest median fitness of solutions with multiple evaluations.

Not all solutions will necessarily finish successfully during an evaluation and therefore will result in no fitness value. These individuals fail the performance functionality predicate *perf*. We will track these occurrences externally to the main evolutionary search in a set *Fail*. However, not all crashes may be repeatable, or common. In order to further examine these instances, an additional operator op_{failed} promotes a solution which failed *perf* to the next generation for re-evaluation. Thus, in each generation, if such an individual exists in *Fail*, we then propagate one into the new generation relative to the rate of *perf* failure. 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 errors in the virtual machine's operation.

C. Fitness Function

The final challenge for the search formulation is that of the fitness function. An individual event sequence *es* is evaluated by executing itself once with the system configured as S^0, without any security mechanism installed, and once as $S^?$, with the security mechanism 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 variable in nature. Instead of considering the first fitness value as final, each time a certain solution is executed its fitness is tracked. The process for selecting parent solutions for an operator is then based on a solution's median fitness over its possibly multiple evaluations. At the same time, extreme values are tracked by the search control to be examined by the user.

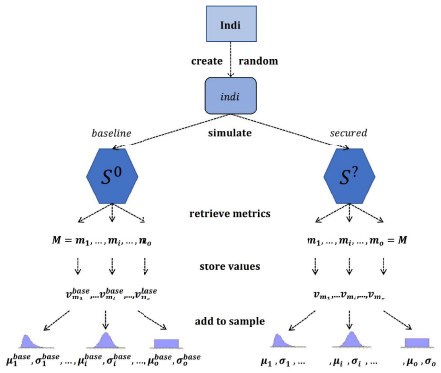

Fig. 1. Metric Sample Distribution Creation

It is important to be aware that the distribution of behaviour for metrics will vary between different interaction event sequences. To counter this, the definition of the differential function *diff* used in the effectiveness measure considers the distribution of each metric. Before the initial population of individuals is evaluated, a sample set of random event sequences is generated and evaluated. The metric values produced during these evaluations 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 secured system. During the following search process, the differential function *diff* references the mean μ and standard deviation σ of the distributions of these samples.

Figure 1 visualizes how the samples for the metric distributions used in the fitness function and differential function are generated.

Definition VII. 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 less desirable values are a larger positive value, $sgn = +1$, effectively the differential value is a comparison like a statistical relative z-score between the two configurations. A z-score is a measure of the standard deviations of an element from the mean of the distribution [6]. 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 $S^?$ increased more than those for S^0. For metrics where worse is a larger negative number, $sgn = -1$, we invert the z-score calculation. As a result, a smaller and more negative z-score for $S^?$ over S^0 will result in larger differential values.

One of the biggest benefits of using a z-score style calculation is that we can normalize the values of metrics which are measured with differing scales. Instead of comparing values directly we 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 twice the value of 1.0, while a weight of 0.5 is half the value.

IV. Exploratory Testing of Anti-Virus Programs

This section describes the experimental design setup. First, the details required to implement the evolutionary algorithm introduced in the previous section will be explained. Next, this paper's experimental configuration for a secured system $S^?$ consisting of the installation of an anti-virus program will be described. Anti-virus programs are a well-known example of a common security mechanism generally developed in isolation which, depending on the usage profile, often produces unwanted performance degradation [4], [7].

The infrastructure requirements for the evolutionary algorithm require the explanation of the code implementation of the search control, as well as a virtual machine configuration. The evaluation of a single individual will require running two virtual machines a length of time which numbers in seconds. Due to the simulation length, optimization of the search control's internal execution has a trivial effect on the overall run time. This fact allows for the inclusion of robust logging and tracking within the search control.

The current search control implementation replaces one generation with a completely new one each iteration. The experimental configuration used for this paper consists of the algorithm implemented in Java. The virtual machines are Windows XP instances operating via VirtualBox 5.1.6. Java and Virtual Box are both platform portable and accessible. Windows XP was chosen due to its reduced space and memory requirements for virtual machine instances. At the same time, in computer science it represents a widely explored and common commercial development platform during the most active period of anti-virus software development.

The baseline unsecured virtual machine S^0 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. The possible interaction programs installed are: Microsoft Excel/Word 2010 to allow document editing actions, VideoLAN VLC player to allow playing of a 480p video file, the Java Development Kit 7 to allow 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 variances of these interactions and classes

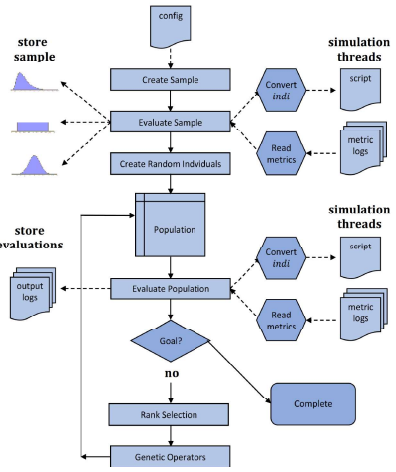

Fig. 2. Search Execution Structure

of interactions exist, or are possible, but were not utilized in this paper's experiments. Each program's unique interactions require the installation of the program and the development of the AutoIT script commands necessary to execute them.

The secured virtual machine $S^?$ consists of the installation of a single anti-virus program. The selection consists of: Avast Free Anti-Virus 17.1.2282, ClamWin 0.98.5, and Ad-Aware Free Anti-Virus 11.8.586.8535. If available, the most robust suite of anti-virus services is enabled upon installation of each. Some of these security mechanisms are expected to have performance challenges. The virtual machines have internet access, therefore to prevent conflicts with definition and program update attempts, the system time and date are detached from that of the host system and made static. The search process of the genetic algorithm is depicted in Figure 2.

Each time an event sequence needs to be evaluated the search control translates it into an AutoIT script. This AutoIT script accumulates the required variable definitions and import statements from all the included programs as declared in each program's definition in the search control implementation. These accumulated script statements form the start of the script. Then from the start of the sequence, each interaction is translated into its AutoIT script and added to the overall script. If the associated program for the interaction has not yet been started, and needs to be, the necessary prior AutoIT script commands are added to launch the program. Additionally, if it is necessary to make the program's window active to perform a command, the necessary AutoIT script commands are added to accomplish this prior to performing the action. Finally, if tracking commands are required such as for the duration of a script or command, then these commands are added as required. The resulting AutoIT script is then stored in a shared folder visible to the virtual machine.

To execute a script on a virtual machine, a snapshot taken after start-up executions have ceased is used for each of S^0 and $S^?$. First, the baseline system S^0 is launched in the snapshot state. Then the script is executed, and the virtual machine is closed. The same series follows for the secured configuration $S^?$. After completion of each of these executions, metrics are compiled from the virtual machine statistical/metric logging and from the script output.

If the script failed to complete, this is generally because either (1) the virtual machine failed to execute the search control's commands successfully, or (2) the script failed to execute the listed interactions successfully. The first is noted as an $error$, the second is noted as $incomplete$. These are both examples of failure under the performance functionality predicate $perf$. However, errors are generally disposable as they are unassociated with the actual event sequence and often a result of the underlying virtual machine infrastructure conflicting with the search control's host operating system.

For the purposes of this paper's experiments, the scripts that completed are evaluated using only the m^{time} metric, which measures the time it took for an experiment to complete. The focus of the paper is to interpret the user's usage profile as an experiment which produces example event sequence. These examples can be evaluated to assist the user in determining a choice between security mechanisms. The metric of time is the most representative of the user's ultimate perception of the system's performance. Other unused metric types, such as hard drive, network, memory, and processor related measures, are available from the virtual machine software. These metrics have more use for security mechanism designers as they focus the exploratory search on specific system resources and behaviour.

For an experimental execution of the evolutionary search, the initial sample size of solutions is 50 and the population size is 25. The operators are chosen 25% mutation and 75% crossover. These parameters are chosen based on recommendations from An Introduction to Genetic Algorithms [5] and from the NIST Handbook [6]. A single solution is propagated in each generation for each of the operators that maintain the best solution and solutions that failed $perf$. During an experiment, individuals that fail the $perf$ predicate are tracked as well as the top event sequences found during the search.

The most fit individuals are tracked according to the largest maximum/median effectiveness measures eff. It should be noted that the top individual from a previous generation will be propagated to the next where it will be executed for another sample of its performance. This results in more comparative data for example cases of bad performance/functionality and confidence in the evaluation. Individual solutions that lack repeatability will be evaluated with a reduced fitness, resulting in other individuals replacing them in future generations.

V. Experimental Results

This experimental results section consists of two subsections. These experiments are a proof of concept that the proposed exploratory search method using evolutionary search for event sequences is applicable to the Effective Security problem. Each of these subsections shares a common experimental configuration of event sequences of length ten. The first subsection examines the execution process of an exploratory search experiment with AdAware installed as a single security mechanism. The other subsection examines the results of an exploratory search designed to assist a user in choosing between the Avast/Ad-Aware anti-virus systems for two different usage profiles. ClamWin is included as a third on-demand only security mechanism for comparison, while both the Avast and Ad-Aware mechanisms involve active monitoring. As already stated, the metric explored and used for the fitness function direction of exploratory tests is the running time required for the execution of the script of event sequence.

The first subsection examines the characteristics of an exploratory search run, including the sample distribution produced for the metric used in the fitness function. The fitness function uses the produced distributions in the differential function $diff$ to create the normalized z-score for a metric's value.

The second subsection explores the top event sequence results produced by exploratory tests running for a hundred generations. Two usage profiles are considered. For each usage profile an experiment for each security mechanisms was completed. From each experiment the top five median fitness individuals were selected. The event sequences forming the DNA of these individuals were extracted to produce an AutoIT script that was re-simulated for each of the three security mechanisms. The resulting information allows us to contrast the effects of the different security mechanisms.

Usage Profile 1 consists of Excel spreadsheet editing, Java JDK compiling, file encryption operations, file copying operations, and IE browser web page interactions. Usage Profile 2 consists of Word document editting, VLC movie/audio playing, FTP transmissions, Zip/Unzip of compressed files and SQLLite database interactions. These two usage profiles were randomly chosen from the ten available programs used in the first experiment. Each experiment took approximately 2-2.5 hours with four virtual machines available on an Intel i7-6850K.

During the completion of these experiments there were examples of event sequences that did exhibit virtual machine management disruption (but no examples of incomplete script execution). Upon re-simulation there was no evidence to support that this behaviour was repeatable and the conclusion was that the captured behaviour was the result of host system scheduling disrupting virtual machine management.

A. Script Time Experiment

When an experiment begins execution it performs a sample evaluation of individuals of the same event sequence length as the experiment. In this experiment the sample size was 50. From this sample, for each metric, a distribution is formed. Figure 3 shows a density plot for the script time distributions produced by this experiment for the unsecured **Without** virtual

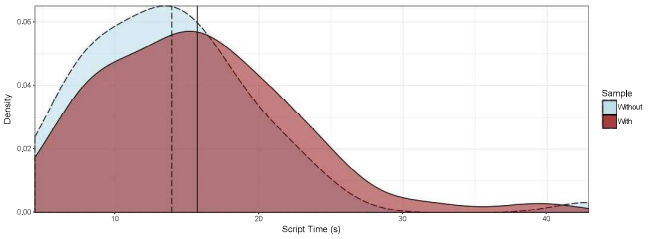

Fig. 3. Script Time Sample Distribution Density

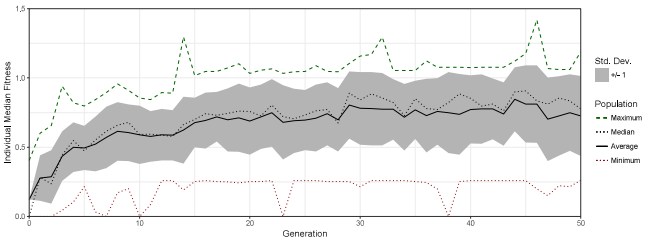

Fig. 4. Population Median Fitness Statistics

machine instance and the AdAware secured **With** virtual machine instance. The mean of each distribution is shown with a line. The expected longer average run time for event sequences once AdAware is installed is in evidence.

An 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. As the experiment continues each individual for a generation is evaluated producing a script time metric for both the secured and unsecured system. In order to direct the experiment's search process these values need to be turned into a fitness value. The experiment uses the distribution from Figure 3 to judge the script time for each configuration relative to its distribution. The resultant z-scores are then compared.

Figure 4 shows population median fitness measures for each generation. It can be noted, that the largest median fitness in a generation spikes each time a new individual with a high fitness is evaluated. However, the largest median fitness in each generation is variable. 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. A search guided by only an outlier maximum fitness from the first time an individual was evaluated could be distracted from its search goal.

B. Usage Profile Security Mechanism Selection

This subsection reports the results of performing an exploratory search experiment for each combination of the three security mechanisms and two usage profiles. The decision support for choosing a security mechanism for each usage profile is based on performing a comparative evaluation of the top five median fitness event sequences produced by each of the three security mechanism tests.

TABLE I
SAMPLE MEAN (ST.DEV.) FOR m^{time} IN S

usage profile	Without	ClamWin	AdAware	Avast
1	18.84 (4.87)	18.38 (4.55)	**28.72 (8.54)**	23.19 (4.99)
2	33.12 (12.65)	33.03 (12.78)	35.78 (13.39)	**38.52 (13.75)**

TABLE II
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)

To begin, Table I reports the sample script time measures for 150 random event sequences evaluated for each of the four virtual machine instances: first the unsecured **Without** virtual machine instance, then a sample for each of the security mechanism installations. **ClamWin** is a passive on-demand security mechanisms so the reported values being roughly equivalent to the **Without** system are expected. Both, **AdAware** and **Avast** are active security mechanisms and result in delays to event sequences run against them.

One experiment was completed for each of the security mechanisms. From each of these, five event sequences with the top median fitness values are selected. Table II and III report the mean and standard deviation of evaluating each of these example event sequences against all four configurations of the **Without**, **ClamWin**, **AdAware**, and **Avast** virtual machine instances.

In Usage Profile 1, **ClamWin** event sequences converged on sequences of primarily file copying actions. Also for Usage Profile 1, the **Avast** and **AdAware** sequences also have file copying, but additionally include more compilation and encryption interactions. The Usage Profile 2 experiments are different as average run time varies between the three different experiments. For Usage Profile 2 the **ClamWin** experiment produced event sequences with many Zip/Unzip actions, the **AdAware** experiment produced event sequences with a number of VLC and Zip/Unzip actions, while the **Avast** experiment produced event sequences with many Word interaction events.

The raw time information reported in Table II and III can be difficult to compare. Instead of raw values, in Figure 5 the top average fitness event sequence from each test for

TABLE III
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)**

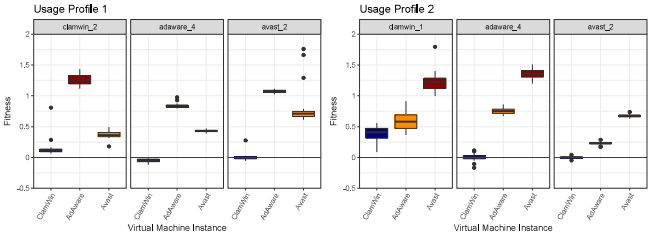

Fig. 5. Top Event Sequence Fitness

the mechanisms has been selected. Instead of reporting the raw script time values the values have been converted into the fitness values the exploratory search system uses and the fitness evaluations depicted as a box and whisker plot.

For Usage Profile 1 the **AdAware** mechanism produced the biggest increase in script time, with **Avast** coming in second rather consistently. However, for Usage Profile 2, the **Avast** mechanisms produced the biggest increase in script time, with **AdAware** coming in second consistently. The conversion also helps demonstrate that between the **Clamwin** and **Without** virtual machine instance all the event sequences are roughly equivalent. This supports the on-demand operation of **ClamWin**.

Based on the results of these two experiments, if the user did not find the run-time of the two on-demand security mechanisms of **AdAware** or **Avast** acceptable, then they could feel safe choosing the **ClamWin** mechanism with no performance concerns. However, this would mean having no active protection. On the other hand, if the performance consequences were acceptable the exploratory search method would recommend **Avast** for Usage Profile 1 and **AdAware** for Usage Profile 2.

VI. RELATED WORK

As already stated in the introduction, the problem of performance losses due to installation of security mechanisms has been ignored in the scientific literature and consequently there have been no decision support tools for it.

Our approach can be classified as a search-based software engineering approach [8], but while in this area there are many works about testing, performance testing is not covered. In addition to the already cited [2], [3], the paper nearest to our approach is [9], which is also concerned with potential performance losses, but due to self-adaptation of a system, so that it did not have to deal with the measurement uncertainty we had to deal with.

VII. CONCLUSIONS AND FUTURE WORK

This paper introduced an evolutionary search method for event sequences designed to explore the performance and functionality consequences resulting from the installation of a security system. The statistical variance and relative comparison characteristics of this effective security problem are an interesting challenge. The exploratory search process is guided by a uniquely developed fitness function that is able to combine, if necessary, a variety of system measures. The fitness function also quantifies the performance and functionality of the secured system relative to that of an unsecured system.

The fitness function comparison developed was accomplished by incorporating the distributions of the metrics. Additionally, the selection process chosen addressed the variance in an individual's fitness between simulations. Proof of concept experimental evaluations were completed to contrast the consequences of installing three different anti-virus security mechanisms for two different usage profiles. The completed experimental evaluation produced event sequence examples that recommended a different security mechanism for each usage profile.

As previously discussed in the paper, the described method is capable of integrating multiple different metrics within the fitness function. Future work will compare multiple different performance metrics pulled from the virtual machine statistics during an event sequence run. At the same time experiments are planned to compare different types of security mechanisms beyond the anti-virus suites compared here.

REFERENCES

[1] M. Silic and A. Back, "Shadow IT - A View from Behind the Curtain," *Computer Security*, vol. 45, pp. 274–283, 2014.

[2] K. P. Bergmann and J. Denzinger, "Testing of Precision Agricultural Networks for Adversary-Induced Problems," in Proc. GECCO'13, 2013, pp. 1421–1428.

[3] T. Flanagan, C. Thornton, and J. Denzinger, "Testing Harbour Patrol and Interception Policies using Particle-Swarm-Based Learning of Cooperative Behavior," in Proc. CISDA'09, 2009, pp. 1–8.

[4] M. E. Locasto, S. Bratus, and B. Schulte, "Bickering In-Depth: Rethinking the Composition of Competing Security Systems," *IEEE Security & Privacy*, vol. 7, pp. 77–81, 2009.

[5] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[6] NIST/SEMATECH, "e-Handbook of Statistical Methods," http://www.itl.nist.gov/div898/handbook/, 2011, [accessed Dec-2018].

[7] Skywing, "What Were They Thinking: Antivirus Software Gone Wrong," *Uninformed*, vol. 4, 2006, [accessed Sep-2018].

[8] P. McMinn. "Search-based software test data generation: A survey," *Software Testing Verification and Reliability*, vol. 14, pp. 105–156, 2004.

[9] J. Hudson, J. Denzinger, H. Kasinger, and B. Bauer, "Efficiency Testing of Self-adapting Systems by Learning of Event Sequences," in Proc. ADAPTIVE-10, 2010, pp. 200–205.