Decision Support for Combining Security Mechanisms using Exploratory Evolutionary Testing

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@cpsc.ucalgary.ca

***Abstract*—We present a process utilizing an evolutionary learning method to explore combinations of security mechanisms with regard to performance problems they might create for a particular user profile. For each combination, the process uses an evolutionary search to identify sequences of interactions with a computer (in form of a virtual machine) that stress the system to a much larger degree with the combination installed than without it. The process then compares the mechanism combinations using the "best sequences" for each combination to suggest the combination that overall has the least impact on performance. The process also explores interaction sequences that caused system failure, or were not able to finish within the given time limit, to identify incompatibilities between security mechanisms. For evaluation, the process was applied to create a tool for finding the best set of multiple anti-virus software systems for Windows XP. In the primary evaluation, the tool identified a set of five mechanisms that did not degrade performance too far, while providing the intended security coverage. At the same time, the tool found a clear incompatibility between two mechanisms as demonstrated by a zip operation failure after only a few interactions.**

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

I. Introduction

The use of a combination of defensive measures to increase the protection of a physical location is a practice that can be observed throughout history. These measures can be of different type, like the moat and walls of a medieval castle, or of the same type, like having several different locks on the door of an apartment. In the literature, this is referred to as *defense-in-depth* (see [7]).

Naturally, combining security mechanism is not only a good idea for physical places, but also digital spaces like computers. However, a user of several computer security measures, trying to achieve *security-in-depth*, faces potential limitations compared to what we see for physical security. As an example, most companies that produce security software recommend not using any security software not produced by them. While some might see this as a mere marketing ploy, there is already evidence of single security mechanisms producing problems for certain user programs (see [3], [8]), due to competition for particular computing resources. With several security mechanisms such resource conflicts are obviously more likely. Naturally, there is no big interest by the developers of security software to provide information on the compatibility of their products with the products of competitors.

Combining compatible security software has the ability to increase the security of a computer. However, the compatibility of security software is often dependent on the particular usage profile of application programs the user interacts with. So far, there is no support for users in deciding if a set of security software can be combined for use within their user profiles. A user has to develop and perform their own tests for a decision.

In this paper, we present a tool and process for users, or the information technology department of a company, to support making such decisions. The tool uses evolutionary learning to identify interaction sequences involving the programs from a user profile. These interaction sequences are examples that could be considered a problem for the user, if a particular combination of security programs is used. The evolutionary learning tool is based at its core on the method presented in [1], but it uses a more complex fitness measure. The tool incorporates this core into a larger process aimed first at the potential users of a combination of security programs, but this process can be extended for security program developers to help them pinpoint problems with other programs.

Our experiments with different anti-virus software on the Windows XP operating system showed that for a given user profile several security software systems can be combined and these combinations compared for selection. We also were able to pinpoint a short interaction sequence with programs within the usage profile that crashed when particular anti-virus software systems were installed together.

This paper is organized as follows: after this introduction, in Section II we formally define the Security-in-Depth problem and the Effective Security-in-Depth problem that we address in this paper. In Section III, we present our extended exploratory evolutionary testing approach for combinations of security mechanisms based on [1] and the process that uses this approach. In Section IV we instantiate the process to anti-virus security mechanisms for Windows XP. We present our experimental evaluation in Section V and Section VI. In Section VII we briefly report on the very sparse related work and in Section VIII we conclude with some remarks on future work.

2375-0197/20/$31.00 ©2020 IEEE
DOI 10.1109/ICTAI50040.2020.00091

II. Definitions

To assist in the definition of the Effective Security-in-Depth (ESiD) problem we will begin by defining the Security-in-Depth (SiD) problem. The SiD problem consists of determining a subset of security mechanisms which achieve a desired depth of coverage, without any regard to the performance of the protected system. The Effective SiD problem additionally limits covering combinations of security mechanisms to those that achieve the desired functionality and minimize unwanted performance consequences. The ESiD problem solutions are therefore a subset of the SiD solutions.

A. The Security-in-Depth Problem

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 $C = \{c_1, c_2, \ldots, c_n\}$. S^{avail} is the set of m available security mechanisms such that $S^{avail} = \{s_1, s_2, \ldots, s_m\}$. Each security mechanism $s_i \in S^{avail}$ is a subset of coverage areas such that $s_i \subseteq C$. These coverage areas have been determined, by either the mechanism's developer or external examination, to be the areas for which the mechanism claims to provide security 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 $c_i \in C$. For example, a depth of $D(c_i) = 0$ indicates that no layers are desired for the coverage area c_i.

Definition I. Coverage Function
The coverage function cov is a function

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

that determines the depth of coverage provided by the set $S^?$ of security mechanisms for each area of coverage $c_i \in C$. When applied to all $c_i \in C$, if

$$cov\left(c_i, S^?\right) \geq D(c_i)$$

then this indicates that $S^?$ provides the desired level of security coverage. $S^?$ is then an element of the covering set of security mechanisms $Cover^D$. □

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

While the SiD problem can become complicated if the coverage subsets of security mechanisms are large, for mechanisms dedicated to cover one or two areas finding a solution is usually not difficult. But the consequences a particular combination $S^?$ has for a user can be rather different, which leads to the need for *effective* combinations.

B. The Effective Security-in-Depth Problem

The ESiD problem extends the base SiD problem to that of selecting a combination of security mechanisms that, in addition to fulfilling the desired depth of coverage, provide the desired functionality and minimizes performance loss. This performance and functionality are evaluated according to metrics observed via interactions with a collection of programs. These metrics and interactions are representative of the performance and functionality loads desired by a user. The Effective Security-in-Depth (ESiD) Problem is a natural extension of the Effective Security (ES) Problem from [1] to considering multiple overlapping (instead of a single) security mechanisms for coverage areas.

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 [3].

An interaction 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 $ev = \left(ip_{j,k}, t\right)$.

ES^P is the set of all possible interaction event sequences given the available interactions I^P. $ES^{eff} \subseteq ES^P$ 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 $sgn = -1 or +1$.

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. A $sgn = N/A$ 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 interactions 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.

Definition III. ESiD Problem

The ESiD problem is the determination of an effective subset S^{eff} of security mechanisms that are a covering subset for the desired depth of coverage D. S^{eff} fulfils a desired performance functionality predicate $perf$ while minimizing the extent of unwanted performance loss in an 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 ES^{eff} which are representative of the user's performance requirements. □

Again, there are likely many possible subsets S^{eff} that fulfil this relation. We define the set $\mathit{Eff}^M \subseteq Cover^D$ as the set of all subsets S^{eff} found in $Cover^D$ that provide the desired performance and functionality under M.

As in [1], the predicate $perf$ is used to decide if a selection of security programs achieves the desired level of performance or functionality in regards 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 mechanisms $S^0 = \{\}$. In other words, S^0 should be such that $perf$ is true for all event sequences in ES^{eff}.

Definition IV. Performance Functionality Predicate ($perf$) The performance predicate $perf$ is a predicate that is true if all observed values v_m of metric $m \in M$ fall within their acceptable range $[a, b]$ such that $a \leq v_m \leq b$, else false. □

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, as in [1], the goal is to avoid event sequences in ES^{eff} which maximize eff. In general, the addition of security mechanisms $S^?$ will have a negative effect on most performance measures we examine 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 in performance for the system with some set $S^?$ of security mechanisms installed and the baseline performance of the system S^0 where none are installed.

Definition V. Effectiveness Measure (eff) The effectiveness measure eff with respect to M and an event sequence $es \in ES^{\mathit{eff}}$ is defined as

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

where $w_i \in \mathbb{R}_{\geq 0}$ and $\mathit{diff}_m\left(v_{m_i}(es), v_{m_i}^{base}(es)\right)$ 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.

III. Decision Support for ESiD

In this section, we present the general concept for a process that provides decision support for the ESiD problem. The process is based on the evolutionary search process for interaction event sequences presented in [1], but it evaluates the event sequences under a combination of security mechanisms and it uses a more complex scheme for fitness functions. In addition to comparing the combination of security mechanisms (that we assume to already solve the SiD problem for the user) using the top event sequences found for the candidate combinations, the process can also be extended to pinpoint incompatibilities of security mechanisms for security system developers. We will start by describing the general process and then cover the evolutionary search approach.

A. Our decision support process

Our process to provide decision support for selecting a combination of security mechanisms solving the ESiD problem is based on comparing the performance of event sequences based on a user profile on two virtual machine instances, one without any security mechanisms (S^0) and one ($S^?$) with a particular combination of mechanisms installed.

Steps: Decision Support

1) Collect the installation software for each of the security mechanisms.
2) Collect the installation software, and accompanying files, for each of the programs from the user's profile.
3) Set up the experimental environment for the evolutionary search tool. Create the virtual machine instance for the fresh installation state without security mechanisms. From this fresh installation, create a virtual machine instance for each of the security mechanism combinations being compared.
4) Create and run an evolutionary search for event sequence experiment for each of these security mechanism combinations.
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 combination that created them.
7) Evaluate the event sequences multiple times for each of the other security mechanism combinations.

8) Compare and contrast the quantitative differences in event sequence performance between the combinations.
9) Select the combination that best fulfils the performance expectations of the intended user.

So, we essentially search for event sequences for each of the candidate security mechanism combinations fulfilling certain conditions (see later), make sure that the issues produced by these sequences are reproducible and then compare the mechanism combinations on all of the so found event sequences. This cross comparison is done multiple times to deal with the fact that two runs of the same sequence are not always leading to the same result for some metrics.

As we will see when looking more closely at the evolutionary search tool, there is the possibility that event sequences are encountered in the experiments that fail the $perf$ predicate. Although these sequences often will be executed again during the search and might then not fail $perf$, our tool nevertheless keeps track of them so that they can be further investigated. This investigation extends the above process as follows:

Additional Steps: Pinpoint Emergent Misbehaviours

1) Revalidate the event sequences by evaluating them multiple times with all the security mechanism combinations giving them more time.
2) Select event sequences that have still too many fails of $perf$, especially those that crash.
3) Shorten the remaining event sequences to the events that lead to crashes.
4) For all events in the sequence, remove it from the sequence and validate the resulting sequence.
5) For all reduced sequences that still crash, repeat the previous step.
6) Repeat the last two steps until a minimal sequence resulting in a fail of $perf$ is found.

In summary, we try to find the shortest possible interaction event sequence that still shows problems with $perf$.

B. The Exploratory Evolutionary Learner

The processes described above are aimed at finding problems with a combination of security mechanisms with regard to a particular user profile. Such a "problem" is an event sequence that demonstrates unwanted behaviour with respect to $perf$ due to the security mechanisms being installed on the virtual machine. Finding these event sequences is a search problem, which is unfortunately not tractable if we allow for sequences of indefinite length. But with a given maximum length the resulting search space is well-suited for evolutionary algorithms, as has been shown in [1]. In the following, we will briefly describe this algorithm with a special focus on the fitness measure used to guide the algorithm towards sequences that stress $perf$.

To represent an individual we should obviously use an interaction event sequence. As a result of having to run such a sequence on two different virtual machines, any more sophisticated timing than just executing the interactions in sequence is not possible, although we add a nop (no-operation) event in I^P to enable delays of a small period of time before proceeding to the next event. This leads us to the following definition of an individual:

Definition VI. 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 of random solutions is of size pop_size. Each random solution is formed from a sequence of random integer pairs. Each pair is constructed by selecting a pseudo-random positive integer for a program in P and an integer for an associated interaction for that program.

As operators we use the following mutation and crossover operators. The search uses a single-point mutation operator mut_{op} and a two-point crossover operator $cross_{op}$. These operators each use a rank-based selection process sel_{proc} to select an individual from the current population [4].

Definition VII. 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 VIII. 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 $indi_{crossover}$. □

In addition to these operators there exist an operator op_{prev_best} to maintain the previous generation's most fit solution. A variant of this 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 the collection $Fail$ of solutions that failed to complete execution, we then propagate one into the new generation relative to the rate of $perf$ failure.

The key element for our exploratory learner is the fitness measure. It obviously has to be based on the metrics in M, more precisely it has to implement eff. But Definition V left out a key component, namely the function $diff$. As already stated, $diff$ is based on measuring a metric twice, once when executing an event sequence es on the machine for S^0 and once on the machine for $S^?$. The fitness is then a measure of the difference in performance between these two executions.

System performance is variable in nature between executions. To avoid having a single evaluation take the place of what is truly a distribution of performance, each time a 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 using the process described previously.

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 to produce a distribution of measurement for each metric. 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.

Definition IX. 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. 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]. 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.

IV. Instantiation to Anti-Virus Software

In this section we instantiate the process described in the previous section to anti-virus security mechanisms operating in a Windows XP environment. 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 [3], [8].

The virtual machines are Windows XP Service Pack 3 instances operating via VirtualBox 5.1.6 and our evolutionary algorithm is implemented in Java. Java and Virtual Box are both platform portable and accessible. Windows XP was chosen due to its reduced space and memory requirements

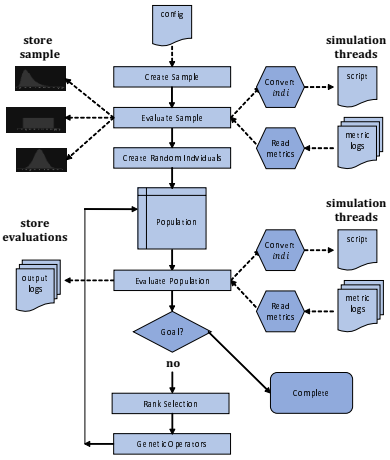

Fig. 1. Search Execution Structure

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. AutoIT 3.3.12.0 was installed to enable automation scripts to be executed to interact with programs on the system. The possible interaction programs installed were: 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. Each program's unique interactions require the installation of the program and the development of the AutoIT script commands necessary to execute them.

The instantiation of the decision support process is depicted in Figure 1. For the fitness measure, we used the following groups of metrics and acceptable ranges: script execution time, a group of hard drive measures, a group of network measures, a group of RAM related measures and finally a group of processor related measures. The acceptable range for script execution time m^{time} was $([0, 5]min, -1)$. The group of hard drive measures consisted of DMA ($sgn = -1$), PIO ($sgn = -1$), number of bytes read ($sgn = -1$) and number of bytes written ($sgn = -1$), all with ranges $([0, +\infty])$ and the indicated sgn. The group of network measures consisted of the number of bytes received ($sgn = -1$), the number of bytes transmitted ($sgn = -1$), the receive rate ($sgn = 1$), and the transmission rate ($sgn = 1$) with ranges $([0, +\infty])$ and the indicated sgn, again. The group of RAM related measures was the number of allocated pages ($sgn = -1$), the RAM cache average ($sgn = 1$), the RAM cache minimum ($sgn = 1$), the average free RAM ($sgn = 1$), and the minimum free RAM ($sgn = 1$) with appropriate ranges (see Section V-A) and the

indicated sgn. Finally, the group of processor related metrics was the average kernel load ($sgn = -1$), the maximum kernel load ($sgn = -1$), the average user load ($sgn = -1$), and the maximum user load ($sgn = -1$), again with appropriate ranges and the indicated sgn. As stated before, the fitness function is a weighted sum of the differential function values over these metrics (using the opposite sgn value since we want to find event sequences that stress the system).

V. Experimental evaluation

This section explores whether the process given in Section III is effective in providing decision support for selecting a subset of security mechanisms providing a given minimal security-in-depth. We will first present the set-up for the experiments, followed by applying the process to determine the best combination of security mechanisms. Finally, we will use the extension of the process to pinpoint a problem with two of the mechanisms.

A. Set-up

In total, we used six well-known security mechanisms in our experiments: **Avast** Free Anti-Virus 18.6.2349, **ClamWin** 0.98.5, **MalwareBytes** 2.1.4.1018, **Panda** Security 15.1.0, **AVG** Anti-Virus Free 17.9.3040, and **AdAware** Free Anti-Virus 11.12.945.9202. **AVG** would only install in a passive state when other security mechanisms were already present on the machine. **AdAware** was initially installed in passive mode but was converted into active coverage after acknowledging the previous installation of other security mechanisms could result in a conflict. All virtual machines had a 20 GB virtual disk with 2 GB of memory.

Each evolutionary search experiment used 50 generations. The initial sample size of solutions for determining the distributions of the values of the different metrics is 50 and the population size is pop_size=25. The operators are chosen 25% mutation and 75% crossover. These parameters are chosen based on recommendations from [4] and [6]. Finally, the weights for the fitness function distributed a value of one among each of the different groups (and assigning a 1 to the differential function for the script execution time). This means, for example, that each metric in the group of hard drive measures got a weight of one quarter.

B. Deciding on a mechanism combination

For this evaluation, we compared three different combinations of anti-virus programs. Combination **Four** consisted of the mechanisms **Avast**, **ClamWin**, **MalwareBytes**, and **Panda**. Combination **Five** added to **Four** **AVG**. Finally, Combination **Six** used all six mechanisms. The search time, using a single virtual machine individual evaluation thread, for these combinations ranged from 2.14, 3.08, to 4.58 days. Each re-evaluation of an example event sequence of emergent misbehaviour for a single virtual machine instance required about half an hour.

Following our process from Section III-A the first step that is something to look at is Step 4, namely looking at the results of the experiments with each combination. Table I presents the 5 best event sequences (by median fitness) for each of the 3 combinations. The last 6 entries in this table are event sequences that were at some point in time put into *Fail* and we will look at these some more in the next subsection. As the table shows, for the combinations **Four** and **Five** a variety of programs are used (and at least for **Four** some convergence has happened), while for combination **Six**, file copying actions, compile actions, and zip actions seem to show a pattern of heavy file access.

TABLE I
Combinations: 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

Again, following our process, we selected the top event sequence for each combination for further evaluation (Step 5) and their median fitness values did not change much, so that we moved them into Step 7 where we compared all combinations with regard to these sequences ($four_1$, $five_1$ and six_1). Figure 2 presents this comparison for the time metric. There is no guarantee that time is the metric 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 **Six** has an obvious impact in slowing down the six_1 event sequence and a noticeable impact on $five_1$.

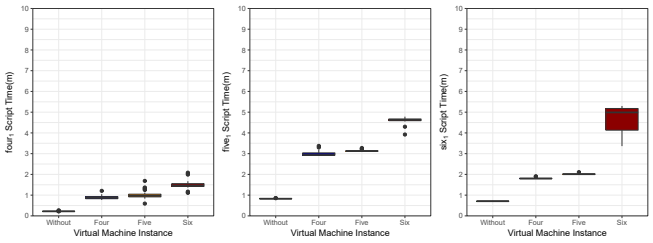

Fig. 2. Combinations: Event Sequence m^{time} Statistics

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 combination **Six**.

Figure 3 shows the comparative comprehensive fitness values for the three example event sequences. 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.

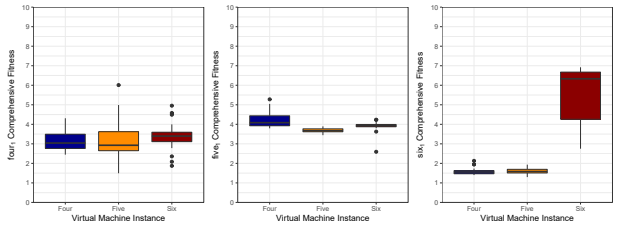

Fig. 3. Combinations: Event Sequence Comprehensive Fitness

If an initial assumption is made that the user has decided that the starting subset **Four** is acceptable, then the reasoning of the process 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.

VI. Pinpointing Emergent Misbehaviour

In this section, we describe how we used the additional steps from Section III-A to pinpoint some problematic behaviour of two of the security mechanisms. For Step 1, we performed re-evaluation tests of twenty-five executions for each of the six examples of prospective emergent misbehaviour from Table I. Each of these examples was an instance of an event sequence that did not complete execution before a time limit of five minutes was reached. 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.

Table II contains the incompletion count for the six examples from the subsets of security mechanism experiments. 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. But increasing the time limit to 10 minutes resulted in no incompletions.

From Table II for Step 2 we determined that only event sequences $four\_inc_1$ and $four\_inc_2$ are of interest for further examination. In other words, Table II 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 working example of incompletion for each of these event sequences.

Table III reports the incompletion counts for reduced sub-sequences (Step 3) taken from event sequence $four\_inc_1$, as evaluated against combination **Four**. First, by examining the

TABLE II
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

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 $four\_inc^5_{1,part}$ (representing Step 4). 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 III
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

After the effort to produce a minimal working sub-sequence example, Table IV re-evaluates this minimal example against all three original combinations of security mechanisms **Four**, **Five**, and **Six**. The incompletion rates were increased in combination **Four** and now occur in all three combinations.

TABLE IV
Minimal Examples Incompletion Count out of 25

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

The hypothesis of why the larger combinations have fewer occurrences of incompletion is that the longer delays found

during the execution of the same sequence of actions on a larger combination allows the file handle to become accessible in time. The issue still occurs in these larger combinations, so the emergent behaviour exists in all the combinations, but the timing of actions that cause its occurrence is less apparent.

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 combination **Three**. The second step was to remove **MalwareBytes** to produce combination **Two**. Then the event sub-sequence was run against **Avast** and **Panda** as individual security mechanisms configurations. Table V 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 any more, due to Windows XP forum support being discontinued.

TABLE V
MINIMAL EXAMPLES INCOMPLETION COUNT OUT OF 25

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

It has been shown previously that not every incompletion is of interest. For example, the event sequence examples produced for combination **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.

VII. RELATED WORK

As already stated in the introduction, there are no works on decision support for choosing security mechanisms that take performance losses and emergent misbehaviour into account, with the obvious exception of [1] for single security mechanisms. Our approach, like [1], can be classified into the area of search-based software engineering (see [5]) and there are many approaches in this area into software testing. But performance testing is not a focus of most of these works and the nearest to what we presented in this paper is [2] which uses evolutionary learning to find problems with self-adapting systems. [2] is only concerned with one performance metric and did not have to deal with the problems of low-level (i.e. virtual machine) variations in metrics.

VIII. CONCLUSION AND FUTURE WORK

We presented a process for evaluating Effective Security-in-Depth in regard to performance using a tool that implements exploratory evolutionary testing to find interaction event sequences from a user profile that stress the system. Additionally, we extended this process to allow pinpointing emergent misbehaviour of combinations of security mechanisms. The completed experiments showed that process and tool performed the intended evaluation and provided decision support. The process was able to recommend that the increase to the Combination **Five** was acceptable, but that the Combination **Six** was not recommended. At the same time the extended process was able to deal with examples of emergent misbehaviour, beyond just simple relative performance considerations.

The experimental evaluations demonstrate 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. The combination of a minimal working example and a highly repeatable failure example, allows 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.

The most immediate future work consists of an exploration of further developing our method to take minimal working examples and use them to create and explore other examples with even more repeatable failure. Another direction for future research is to use our process with other security mechanisms that provide coverage other than anti-virus protection and for other operating systems.

REFERENCES

[1] J. Hudson and J. Denzinger, "Using Exploratory Testing for Decision Support in Choosing a Security Mechanism," in Proc. CEC 2019, pp. 2237–2244, 2019.

[2] 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.

[3] 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.

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

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

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

[7] M. Rosenquist, "Defense in Depth Strategy Optimizes Security," Tech. rep Intel, 2008.

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