Relating the Empirical Foundations of Attack Generation and Vulnerability Discovery

Tyler Westland, *University of Cincinnati*, Cincinnati, OH, USA, westlatr@mail.uc.edu
Nan Niu, *University of Cincinnati*, Cincinnati, OH, USA, nan.niu@uc.edu
Rashmi Jha, *University of Cincinnati*, Cincinnati, OH, USA, rashmi.jha@uc.edu
David Kapp, *AFRL/RYWA*, Dayton, OH, USA, david.kapp@us.af.mil
Temesguen Kebede, *AFRL/RYWA*, Dayton, OH, USA, temesgen.kebede.1@us.af.mil

***Abstract*—Automatically generating exploits for attacks receives much attention in security testing and auditing. However, little is known about the continuous effect of automatic attack generation and detection. In this paper, we develop an analytic model to understand the cost-benefit tradeoffs in light of the process of vulnerability discovery. We develop a three-phased model, suggesting that the cumulative malware detection has a productive period before the rate of gain flattens. As the detection mechanisms co-evolve, the gain will likely increase. We evaluate our analytic model by using an anti-virus tool to detect the thousands of Trojans automatically created. The anti-virus scanning results over five months show the validity of the model and point out future research directions.**

***Index Terms*—Software security, malware detection, automatic exploit generation, anti-virus, information foraging**

I. Introduction

Creating exploits for attacks like control flow hijacking is typically a manual process that requires security expertise [1]. To improve efficiency and increase wider use of exploits in security testing and auditing, researchers have proposed approaches to automatically generating attacks. In web applications, for example, Kiezun *et al.* [2] introduced the ARDILLA tool for devising SQL injection and cross-site scripting (XSS) attacks based on dynamic taint propagation and input mutation. Attacks on binary programs have also been automatically generated. Huang *et al.* [3] investigated software crashes and performed concolic executions by following the failure directed path. Their approach could generate exploits of such vulnerabilities as stack and heap overflows.

Not only are attack vectors generated, malicious programs like worms, viruses, and Trojans are also created. A class of methods focuses on the automatic generation of malware signatures, e.g., DeepSign [4], Polygraph [5], and TrustSign [6]. Another group employs the mechanism of metamorphic malware where the malware would be able to transform its own code so as to create variants of itself [7]. To reduce the number of possible variants to be recognized, normalizer is needed. Researchers have built different normalizers based on term rewriting [8] and other means.

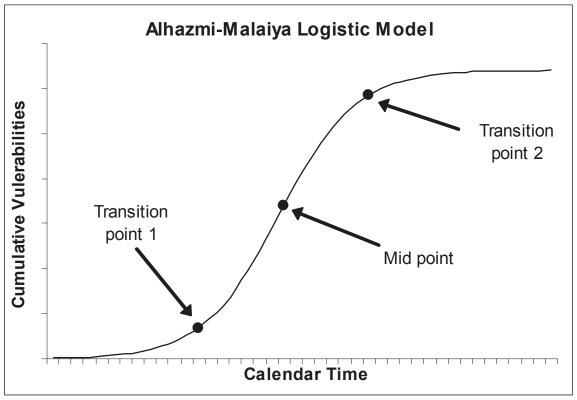
Despite these advances, the empirical base of automatic attack generation is built mainly on studying individual subject systems and their specific implementations. ARDILLA, for instance, was evaluated on five open-source programs, ranging from 326 to 8,181 lines of code, where the tool found a total of 68 attack vectors in these programs with an average of 14% false-positive rate across the three vulnerability types: SQL injection, first-order XSS, and second-order XSS [2]. Others extended such empirical findings. Notably, Huang *et al.* [9] improved ARDILLA's web platform independence and showed their resulting tool's success in generating attacks against four of the five PHP programs assessed in [2] as well as additional ones written in Python.

Evaluating individual systems presents isolated pieces of evidence; however, little is known about *sustained* effect of automatic attack generation. In this paper, we bridge the gap by relating attack generation and vulnerability discovery through a cost-benefit perspective. Vulnerability discovery is chiefly concerned with finding vulnerabilities on a continuous basis yet within finite, economic considerations. Oftentimes, the development team uses a combination of techniques (penetration testing, static analysis, code reviews, etc.) throughout the software development lifecycle while the effort expended and the benefit gained must be cautiously monitored [10].

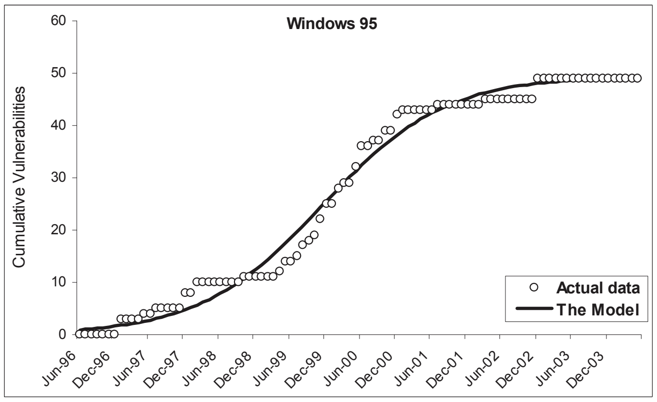
A foundational model of vulnerability discovery is developed by Alhazmi and Malaiya [11], known as the Alhazmi-Malaiya Logistic (AML) model. The AML model depicts how the cumulative vulnerabilities discovered correspond to the calendar time. Empirically, this model has shown to fit the real-world vulnerability data sets, acting as an important conceptual reference for understanding the practicalities of the process of vulnerability discovery. We develop mappings between the AML model and attack generation, and further treat a free, evolving anti-virus (AV) tool as an integral part in our work to establish the practical interests.

This paper makes three main contributions: linking the cost-gain dimensions of attack generation to those of the AML model, articulating our novel model in the context of Trojan creation, and confronting our model with empirical data collected from over five months' scanning of a state-of-the-practice AV tool. The rest of the paper is structured as follows: Section II provides the background information about the AML model, Section III maps the key constructs of the AML model to attack generation, Section IV presents the empirical evaluations of our model, Section V compares related work, and finally, Section VI draws concluding remarks and outlines future work.

978-1-7281-1054-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IRI49571.2020.00014



(a)



(b)

Fig. 1. (a) AML model of the vulnerability discovery process, and (b) AML model fit to Windows 95 vulnerability data set (*both adapted from [12]*).

II. AML Model of Vulnerability Discovery

In his book, *Software Security: Building Security In*, McGraw [13] draws on his experience as a security researcher and claims: "Security problems evolve, grow, and mutate, just like species on a continent. No one technique or set of rules will ever perfectly detect all security vulnerabilities." Austin and Williams [10] substantiated McGraw's claim with empirical evidence where they conducted a case study on two electronic health record systems and compared four vulnerability discovery techniques: systematic and exploratory manual penetration testing, static analysis, and automated penetration testing. The results clearly showed that no single technique detected every type of vulnerability, positioning vulnerability discovery in the empirical ground where, in order to choose the appropriate techniques, expectations of cost and benefit must be established.

Alhazmi and Malaiya [11] developed a logistic model of vulnerability discovery. Figure 1 shows the AML model. The vulnerability discovery rate increases at the beginning, reaches a steady rate, and then begins to decline. The cumulative number of vulnerabilities thus shows an increasing rate at the beginning as the system starts attracting its use base. After some time, a steady rate of vulnerability finding yields a linear curve. Eventually, as the vulnerability discovery rate starts dropping, there is saturation due both to reduced attention and a smaller pool of remaining vulnerabilities [12].

The model assumes that the vulnerability discovery rate is given by the differential equation:

$$\frac{d\Omega}{dt} = A\Omega(B - \Omega) \tag{1}$$

where Ω is the cumulative number of vulnerabilities, t is the calendar time (t=0 initially), and A and B are empirical constants. The rate of change, $\frac{d\Omega}{dt}$ of equation (1), is governed by two factors. The first factor, $A\Omega$, increases with the time needed to take into account the rising share of the use base. The second factor, $(B - \Omega)$, declines as the number of remaining undetected vulnerabilities declines.

While it is possible to obtain more complex models, the AML model provides a good fit to the real-world data, e.g., Figure 1b shows the model fit to the actual data for cumulative number vulnerabilities Ω for Windows 95 [12]. By solving the differential equation (1), one obtains:

$$\Omega(t) = \frac{B}{BCe^{-ABt} + 1} \tag{2}$$

where C is the integration constant. It is thus a three-parameter model given by the logistic function. In equation (2), as t approaches longer discovery period, Ω approaches B. Therefore, the parameter B represents the total number of accumulated vulnerabilities that will eventually be found. It should be noted that the saturation phase may not be seen in an operating system which has not been present for a sufficiently long time. In addition, if the initial adaptation is quick due to better prior publicity, in some cases the early learning phase (when the slope rises gradually) may not be significant [12].

Not only is the AML model fit to the actual data, it also provides conceptually simple understanding of the tradeoffs during the vulnerability discovery process. For example, the curve of Figure 1a bends at two *transition points* and one can further identify these points by taking the derivatives of equation (2) with respect to t [12]. From t=0 to the first transition point, the rate of vulnerability discovery grows slow, indicating that the development team has secured a large amount of typical use cases of the software (e.g., Windows 95) before releasing it.

The rate of vulnerability discovery becomes larger after the first transition point, implying security problems emerge with a close to linear trend. Such a trend diminishes after the second transition point. The highest vulnerability discovery rate occurs at the midpoint of Figure 1a:

$$T_m = \frac{-\ln\left[\frac{1}{BC}\right]}{AB} \tag{3}$$

When t passes the second transition point, the AML curve flattens; however, it is important to point out that certain vulnerabilities remain undetected. Clearly, the AML model makes simplified assumptions, such as the software (especially the security of the software) is thoroughly tested before the release and the no effect of continuously releasing (e.g., patching) is taken into account. Nevertheless, the AML model is useful for vulnerability discovery, e.g., one can use the data from prior and relevant software systems (e.g., operating systems [12]) to constrain the range of the regression parameters' values thereby building an estimate of the duration between the two transition points.

III. Modeling the Detection of Malware Generation

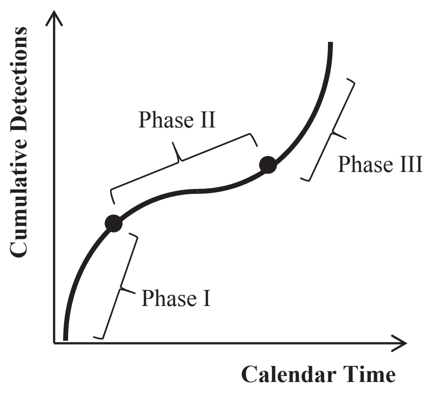
Inspired by the AML model of vulnerability discovery, we develop an analytical model in the context of attack generation. In particular, the attacks that we consider are malware instances generated automatically. For example, a metamorphic engine performs program transformations where a payload is applied to the source program [8]. A single engine, therefore, can be attached with a variety of payloads. This can lead to a massive number of malicious programs.

Although creating malware may reach a high level of automation, detecting the generated malware resembles vulnerability discovery. In both situations, calendar time can be regarded as the surrogate for *cost*. The *gain* can be measured by cumulative detections in essentially the same manner as cumulative vulnerabilities in the AML model. The key difference is that the to be discovered vulnerabilities are unknown at t=0, but all the attacks (malware) are known prior to t=0. For this reason, malware detection exhibits the inverse trends of vulnerability discovery.

Figure 2a shows our malware detection model in which the rate of gain is contrary to that in Figure 1a. Three phases are distinguished, and their contrasts are summarized in Figure 2. When state-of-the-art detection mechanisms (e.g., AV tools) are employed, we expect the initial phase to be a productive period, i.e., a sizable malware would be detected. This is because, for instance, good metamorphic engines are known to be difficult to create [8]. State-of-the-practice AV tools, thus, are quite capable of detecting a fair proportion of the automatically generated malware with little or no time.

Phase II in the AML model is about identifying new vulnerabilities. Correspondingly, the second phase of malware detection is about responding to the survived (undetected) instances after the initially productive period. As shown in Figure 2a, this middle phase can experience a flat curve, implying that the survived malware may continue to live (remain undetected) for some period of time. However, if scanning with the survived sample on a continuous basis, the detection mechanisms co-evolve [7]. This would lead to a nontrivial proportion of the malware being detected, as shown in Phase III of Figure 2a.

While the intended use of our malware detection model depicted in Figure 2a is to allow for empirically sound estimations and predictions, this model currently serves as an analytic tool for understanding the sustained effect of automatic attack generation and the cost-benefit tradeoffs of detection throughout an extended period of time. To lend strengths of this analytic model, we next present some preliminary results of creating Trojans and detecting them via a state-of-the-practice AV tool.



(a)

	AML model	Model of Fig. 2a
Phase I	slower growth; less productive vulnera-biity discovery period	faster growth; more productive malware detection period
Phase II	faster growth; new vulnerabilities emerge	slower growth; malware detection stalls
Phase III	vulnerability discovery reaches saturation	detection mechanisms co-evolve

(b)

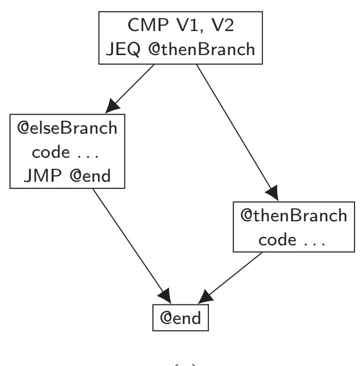
Fig. 2. (a) Modeling the detection of automatically generated attacks (malware), and (b) linking the constructs between the models.

IV. Trojan Detection via VirusTotal

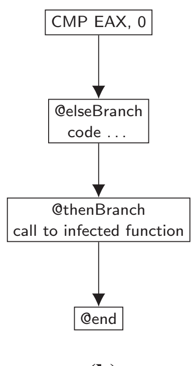
A. Trojan Creation

A particularly interesting type of malware is the Trojan. A Trojan is a benign program that has within it a functional piece of malware, known as a payload. The Trojan must still be usable by a user for all or most of its original uses, but must also consistently execute the payload. To formalize this, the Trojan must pass all or most of the test cases created by the developers of the benign software, as well as a separate set of test cases for the payload's functionality.

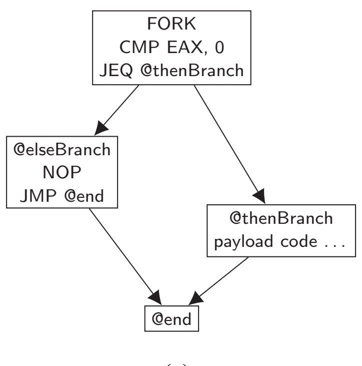
In this work, we take advantage of the Metasploit repository of payloads (https://www.metasploit.com). To mitigate directly using these potentially well-known payloads, we generate variations by implementing ghost writing that swaps assembly instructions with equivalent instructions. Our method of inserting payloads works via replacement instead of addition. This means that in order to insert our payload we must identify parts of the benign program to replace. This is the core concept we adopt from [14]. A payload is usually relatively small and



(a)



(b)



(c)

Fig. 3. (a) Abstraction of a typical if-then-else in assembly, (b) abstraction of an infected if-then-else in assembly, and (c) abstraction of the 'fork' wrapper around a payload.

is no larger than typical functions. To insert a payload we first replace the assembly code within a function with 'NOP' instructions. These instructions are essentially space filler as they inform the computer to take no actions. We then insert the payload into the function via replacement. To ensure that functionality is preserved a function that is either never used or unlikely to be used should be chosen.

To execute our payload we next identify an if statement with the following properties: will always be executed, and has a branch with a low probability of being chosen. Figure 3a shows a pseudo code version of an if statement in assembly code broken into basic blocks. A basic block is a sequence of instructions ending in an instruction that changes instruction flow (e.g. JEQ, JMP, RET). We want to replace one of the branches with a call to our infected function. We also want the control flow instructions to be removed such that the unmodified branch is always executed. Figure 3b shows the simple replacement that occurs. Our Trojan generator is able to identify the start and end of the branches if given the location of the first jump address via information exposed with radare2 (https://github.com/radare/radare2). The result is that the benign program will, for example, always execute the code in the else branch and call the infected function with our payload. This alone does not ensure that functionality is intact as the payload may run indefinitely, thus resulting in the rest of the benign program not executing.

To ensure that a payload does not prevent a benign program from finishing its execution, we execute the payload within a new thread. The code for starting a new thread in the Linux environment is fairly simple with the system call 'fork'. The 'fork' command will start a new thread and allow each thread to identify themselves as the new or old one via a value in the EAX register. This allows one to have the new thread execute the payload while the old thread jumps over it and exits the infected function, as shown in Figure 3c.

Our mutations were done exclusively on the payload portion of the Trojans, not including the fork. Our first method was to remove the error checking of the payload. The error checking served to retry connecting to the attacker's machine if it failed.

```
mov eax, 1 // eax -> 1
mov ebx, 0 // ebx -> 0
int 80h // Exit 0
```

(a)

```
xor eax, eax // eax -> 0
inc eax // eax += 1
xor ebx, ebx // ebx -> 0
int 80h // Exit 0
```

(b)

```
mov eax, 1 // eax -> 1
mov ecx, 1 // ecx -> 1
mov ebx, 0 // ebx -> 0
mov edx, 1 // edx -> 1
int 80h // Exit 0
```

(c)

```
mov eax, 2 // eax -> 2
sub eax, 1 // eax -> 1
mov ebx, eax // ebx -> 1
dec ebx // ebx -> 0
int 80h // Exit 0
```

(d)

Fig. 4. (a) A simple 'exit 0', (b) a clever 'exit 0', (c) a wasteful 'exit 0', and (d) a convoluted 'exit 0'.

Without this it would only try to connect once, but is still not something that can be considered benign. We performed removal in two ways: first by replacing instructions with NOP instructions such that the size of payload is maintained, and secondly by deleting the instructions such that the payload shrank in size. We then tried deleting instructions from the end of the payload. These instructions were responsible for cleanly ending the program with an 'exit 0' after the attacker closes the connection. Without this the program might crash. We made several versions with inserted instructions that would not affect functionality and replacing instructions with equivalents like those shown in Figure 4.

TABLE I
CHARACTERISTICS OF THE EXPERIMENTAL PROGRAMS AND TROJANS

Program	Functionality	Testing	# of SLoC*	ave. cyclomatic complexity	# of Trojans	scanning periods
Random String (RS)	Prints "EVEN" or "ODD" or with equal chance	Tests that either permissible string is printed	75	1.3	9,048	Jan. 15, 2020 – June 14, 2020
Random Adventure (RA)	Game with 3 heroes & 2 monsters (attacked by "sword" or "magic")	Tests that all heroes can win with only "sword" & 2/3 heroes can win with only "magic"	180	2.6	7,852	May 13, 2020 – June 14, 2020

*SLoC: source lines of code

B. Experiment Setup

We created two C programs for evaluating the analytic model of Figure 2a. Table I lists some basic information of the benign programs: Random String (RS) and Random Adventure (RA). Both are small and simple programs as manifested by the source lines of code and the average cyclomatic complexity values. Compared to RS, RA is slightly larger and more complex.

We began the experimental work with RS. The AV scanning of RS, as shown in Table I, was ahead of that of RA. Figure 5 shows the code snippets of RS. For space reasons, source code of RA is not displayed. Figure 5a illustrates that the code that fills the character array is created in the main function. The character array will be filled with either "ODD string \n" or "EVEN string \n" with a 50% chance for both. The if statement in the random_string function of Figure 5b is the one that we target with our Trojan insertion. Our Trojan creation mechanism is able to manipulate either the then or else branch, but our samples all manipulate the then branch. This means that our samples all print "EVEN string \n" and run the inserted payload.

To make inserting payloads simple, we wanted to eliminate the concern about limited space within the benign program. Figure 5c shows the unused_function within RS. This function has the statement "i += 45;" occur 31 times within it. This results in the function being exceptionally large within RS. This function is also never used within the benign sample so any modification to it will not corrupt the intended behavior.

Our benign sample was compiled for 32bit Linux with debug symbols and dynamically linked. The tests of RS are functional, "Does the program print either "EVEN" or "ODD" strings?" To test the payload portion of our Trojans we ran Metasploit to accept connections from a running payload. After a successful connection, we tested basic features like creating a file on the victim's system. This is sufficient as all features of the Metasploit shell are downloaded from the Metasploit instance upon connection. Following the mechanism presented in Section IV-A, a total of 9,048 and 7,852 Trojans are created based on RS and RA respectively.

As these Trojans are created in an automated way, the evaluation of our analytic model lies in the detection of Trojans. To this end, we adopt a free online AV tool, VirusTotal (https://www.virustotal.com/), which has been used as a benchmark in malware detection [14]–[16]. Due to the VirusTotal academic APIs' constraint of 10,000 samples per day, we performed the scanning of Trojans in 3 to 4 batches daily. Figure 6 shows a sample script that we wrote to collect the VirusTotal scanning results. We prioritized the Trojan samples based on their last scan time, e.g., Trojans which have not been scanned at all will be given the highest priority.

```
int main(int argc, char *argv[]){
    char string[50];
    random_string(string);
    printf("%s\n", string);
    return 0;
}
```

(a)

```
void random_string(char *string){
   srand(time(NULL));
   int choice = rand()%2;

   if (choice == 1) {
     strcpy(string, "ODD string\n");
   } else {
     strcpy(string, "EVEN string\n");
   }
}
```

(b)

```
void unused_function() {
  int i = 0;
  i += 45;
  // ...
  // i += 45 occurs 30 times
  printf("%d\n", ++i);
}
```

(c)

Fig. 5. (a) The main function of RS, (b) the random_string function of RS, and (c) the unused_function of RS.

C. Results and Analysis

The VirusTotal scanning results are plotted in Figure 7. We calculate the detection ratio as the number of scans that detected a Trojan to be a virus over the number of scans that were run. The number of scans run includes scans that timed out before concluding if the sample is malicious. In Figure 7, cumulative detection ratios are plotted where only non-decreasing ones are considered.

The scanning period of RS lasts for five months. From Figure 7a, we could recognize a couple of cycles of Figure 2a. The first cycle reaches the end of Phase I in about a month with a cumulative detection ratio of about 0.07. For the next

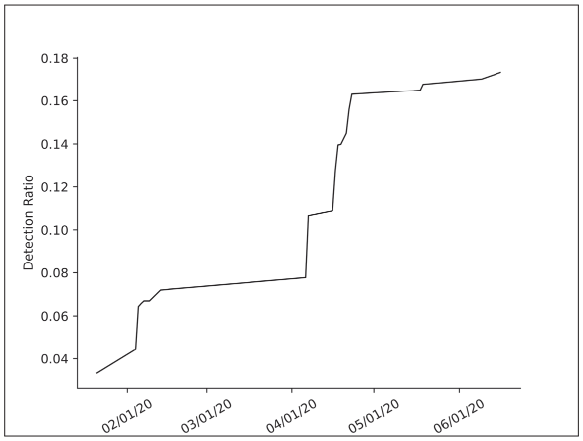
```
# Create VirusTotal API object
vtotal =
      virustotal3.core.Files(VIRUSTOTAL_API_KEY)
# First scan?
if len(sample.virustotal_reports) == 0:
    response = vtotal.upload(sample.file_path)
# Rescan this file
else:
    response =
          vtotal.analyse_file(sample.sha256)

if not quiet:
    print("Analysis Id:
          {}".format(response['data']['id']))

# Return result from adding report
return add_report(str(response['data']['id']),
        wait_time=wait_time,
         max_number_of_attempts=max_number_of_attempts,
                quiet=quiet,
         session=session)
```

Fig. 6. Sample script for collecting VirusTotal scanning results.

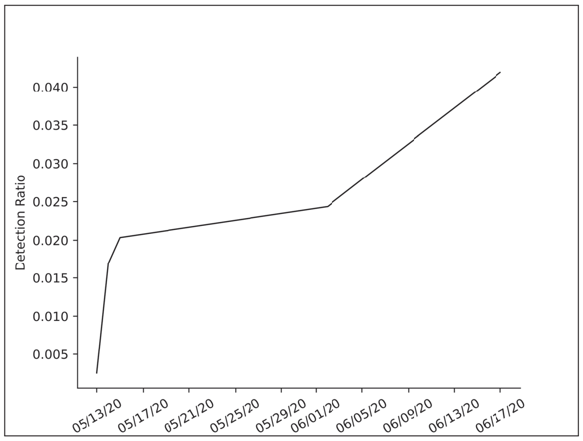
month and a half, the ratio stays smooth, signaling Phase II where a stalled malware detection of the AV tool is observed. For reasons like AV tool's co-evolution, a sudden and sharp increase in Trojan detection appears in Phase III. If we regard this Phase III of the first cycle as Phase I of the second cycle, then a new Phase II seems to emerge and continue till the end of our scanning period of RS.

While Figure 7a of RS depicts five-month data, Figure 7b of RA provides a finer-grained, zoomed-in view. The three phases of Figure 2a become more prominent in the results of RA. Not only is Phase I shorter in Figure 7b, but it is smoother than Figure 7a. This could be explained by the "cold start" of VirusTotal in scanning RS's Trojan samples; however, when similar kinds of samples carried by RA are scanned, VirusTotal has bypassed the "cold start". Nevertheless, the detection ratio growth from 0.020 remains flat for the Phase II of Figure 7b before a linear Phase III is observed.

It is important to point out that certain Trojans in both RS and RA remain undetected by VirusTotal, although most malware samples created by us have been detected. Relating to the AML model of Figure 1a, this implies that no discovery technique or techniques would uncover all the vulnerabilities, confirming the empirical findings of Austin and Williams [10]. In an attempt to build further connections of our results with the AML model, we fit the data of Figure 7 by using least-squares with the three free parameters [12]. The free parameters, together with χ^2 values, are shown in Table II. These inferential statistics show that our model of malware detection is not yet quantitative or predictive, but is currently analytic and qualitative. Next we discuss the limitations of our study so as to enable further quantitative and predictive investigations of our model.



(a)



(b)

Fig. 7. (a) Trojan detection of RS, and (b) Trojan detection of RA.

TABLE II
FIT USING LEAST-SQUARES

	A	B	C	χ^2	χ^2 critical	p-value
RS	0.01	0.213	17.55	0.040	7.81	1.00
RA	0.01	0.558	77.45	0.013	7.81	0.91

D. Threats to Validity

One construct validity of our current study is measuring the cumulative benefit by AV detection ratio rather than by the raw number of Trojans detected. A main reason of our ratio-based measure is due to the daily scanning limits of VirusTotal. The total number of scans, therefore, varies from one batch to another. A threat to internal validity relates to our AV scanning strategy which sets the priority based on the last time a particular Trojan was scanned. For reasons like timeout, other scanning priorities may result in different cost-benefit patterns.

Several factors affect the external validity. Both our benign programs are purposefully simple and easy to insert a payload into. Therefore, the preliminary results reported here are initial findings in relating the AML model to attack generation and detection. Moreover, the Metasploit payloads are well-known; however, even with these payloads, a state-of-the-practice AV tool was shown to go through distinct phases in malware detection. The use of VirusTotal presents another threat to generalizability. While it is interesting to employ other AV tools like TrendMicro, building an infrastructure supporting continuous malware scanning is important to understand the sustained impacts and tradeoffs of attack generation and detection.

V. Related Work

We have drawn close parallels between the AML model of vulnerability discovery and our empirical model of malware detection. In this section, we further shed light on the cost-benefit tradeoffs in software engineers' information foraging tasks where human-tool integration is important. In software development, many information-intensive tasks exist, such as vulnerability discovery and requirements tracing. In essence, these tasks require humans—sometimes assisted by tools—to locate the relevant information (e.g., a bug in the implementation that has security implications or a code fragment satisfying a requirement [17], [18]).

Software developers seeking information adopt various strategies, sometimes with striking parallels to those of animal foragers [19]. Optimality here refers to the strategy that maximizes the gain per unit cost. Figure 8a illustrates the patch model of optimal foraging by presenting a hypothetical bird seeking food in an environment that consists of patches of berry clusters. The forager must expend some amount of *between-patch* time (t_B) arriving at the next patch. Once in a patch, the forager faces the decision of keeping *within-patch* foraging (t_W) or leaving to seek a new patch. As the forager gains energy, the amount of food diminishes or depletes. In such cases, there will be a point at which the expected future gains from foraging within a current patch diminish to the point that they are less than the expected gains that could be made by leaving for a new one. Figure 8b shows Charnov's Marginal Value Theorem, which mathematically models an optimal forager's time allocation. In Figure 8b, $g(t_W)$ represents a decelerating expected net gain function. The amount of energy gained per unit time of foraging is $R = g(t_W)/(t_B + t_W)$. Thus, the rate-maximizing time, t^*, occurs when the derivative of $g(t_W)$ is equal to the slope of the tangent line R^*.

Although the theoretical foundations like the patch model of Figure 8 helps to unify observations (even seemingly conflicting ones) [21] and build tool support in a principled manner [22]–[26], some simplifying assumptions are made in the model. Similar to recent work [27], our work examines the cumulative growth of gain on a cost scale. Different from [27], we use the AML model to depict different phases and present initial empirical data in malware detection. Interestingly, Hussein *et al.* [28] modeled attackers and their intentional goals with constructs adopted from social information foraging [29].

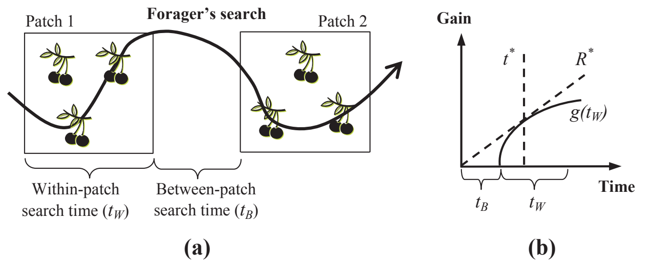


Fig. 8. **(a)** Illustration of patchy environment, where a hypothetical bird forages in patches containing berry clusters. **(b)** Charnov's Marginal Value Theorem states that the rate-maximizing time to spend in patch, t^*, occurs when the slope of the within-patch gain function $g(t_W)$ is equal to the average rate of gain, which is the slope of the tangent line R^* (*both adapted from [20]*).

The extension along the social dimensions has already inspired new ways to construct an information patch [30] and to answer practitioners' requirements tracing questions [31], [32]. It likely will offer insights into synthesizing tools (e.g., Trojan creation and AV) to achieve complementary effects [33] and to assuring a system's safety, security, dependability and other critical properties [34]–[38].

VI. Conclusions

This paper presents our view of attack creation and detection in relation to the AML model of vulnerability discovery. We develop a three-phase model to suggest that the cumulative malware detection has a productive period before the rate of gain slows down. As the detection mechanisms co-evolve, the gain will likely increase; however, relying on single mechanism or mechanisms may not be able to detect all the malware samples. To test our analytic model, we automatically created Trojans on top of two benign programs and further used a state-of-the-practice AV tool to detect the Trojans. The preliminary results confirm the three-phased model; however, the statistical inferences indicate more work shall be done to make the model a better fit.

The future work thus includes calibrating the model and conducting more empirical studies to confront the model with real-world malware detection data. Furthermore, we plan to build different Trojan creation methods (e.g., using payloads other than those from Metasploit), and to develop automated ways to identify the locations in the benign program to insert the payloads. Finally, we encourage replications, especially theoretical replications with open data [39]–[41], of our work so as to offer explanatory powers and to enrich the empirical bases of attack generation and detection.

Acknowledgment

Funding is provided by the AFRL (Air Force Research Lab) via the DAGSI (Defense Associated Graduate Student Innovators) program of SOCHE (Southwestern Ohio Council for Higher Education): Project number RY7-UC-19-1.

References

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[2] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009, pp. 199–209.

[3] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai, "Software crash analysis for automatic exploit generation on binary programs," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 270–289, March 2014.

[4] O. E. David and N. S. Netanyahu, "DeepSign: Deep learning for automatic malware signature generation and classification," in *International Joint Conference on Neural Networks (IJCNN)*, Killarney, Ireland, July 2015, pp. 1–8.

[5] J. Newsome, B. Karp, and D. X. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, May 2005, pp. 226–241.

[6] D. Nahmias, A. Cohen, N. Nissim, and Y. Elovici, "TrustSign: Trusted malware signature generation in private clouds using deep feature transfer learning," in *International Joint Conference on Neural Networks (IJCNN)*, Budapest, Hungary, July 2019, pp. 1–8.

[7] C. Nachenberg, "Computer virus-antivirus coevolution," *Communications of the ACM*, vol. 40, no. 1, pp. 46–51, January 1997.

[8] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia, "Normalizing metamorphic malware using term rewriting," in *International Workshop on Source Code Analysis and Manipulation (SCAM)*, Philadelphia, PA, USA, September 2006, pp. 75–84.

[9] S.-K. Huang, H.-L. Lu, W.-M. Leong, and H. Liu, "CRAXweb: Automatic web application testing and attack generation," in *International Conference on Software Security and Reliability (SERE)*, Gaithersburg, MD, USA, June 2013, pp. 208–217.

[10] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Banff, Canada, September 2011, pp. 97–106.

[11] O. H. Alhazmi and Y. K. Malaiya, "Quantitative vulnerability assessment of systems software," in *Annual Reliability and Maintainability Symposium (RAMS)*, Alexandria, VA, USA, January 2005, pp. 615–620.

[12] ——, "Prediction capabilities of vulnerability discovery models," in *Annual Reliability and Maintainability Symposium (RAMS)*, Newport Beach, CA, USA, January 2006, pp. 86–91.

[13] G. McGraw, *Software Security: Building Security In*. Addison-Wesley, 2006.

[14] E. Sanchez, M. Schillaci, and G. Squillero, *Evolutionary Optimization: The μGP Toolkit*. Springer, 2011.

[15] H. Huang, C. Zheng, J. Zeng, W. Zhou, S. Zhu, P. Liu, S. Chari, and C. Zhang, "Android malware development on public malware scanning platforms: A large-scale data-driven study," in *International Conference on Big Data (BigData)*, Washington, DC, USA, December 2016, pp. 1090–1099.

[16] C.-T. D. Lo, P. Ordóñez, and C. C. Mora, "Towards an effective and efficient malware detection system," in *International Conference on Big Data (BigData)*, Washington, DC, USA, December 2016, pp. 3648–3655.

[17] A. Sardinha, Y. Yu, N. Niu, and A. Rashid, "EA-tracer: Identifying traceability links between code aspects and early aspects," in *ACM Symposium on Applied Computing (SAC)*, Trento, Italy, March 2012, pp. 1035–1042.

[18] N. Niu, W. Wang, and A. Gupta, "Gray links in the use of requirements traceability," in *International Symposium on Foundations of Software Engineering (FSE)*, Seattle, WA, USA, November 2016, pp. 384–395.

[19] N. Niu, A. Mahmoud, and G. Bradshaw, "Information foraging as a foundation for code navigation," in *International Conference on Software Engineering (ICSE)*, Honolulu, HI, USA, May 2011, pp. 816–819.

[20] N. Niu, X. Jin, Z. Niu, J.-R. C. Cheng, L. Li, and M. Y. Kataev, "A clustering-based approach to enriching code foraging environment," *IEEE Transactions on Cybernetics*, vol. 46, no. 9, pp. 1962–1973, September 2016.

[21] N. Niu, A. Mahmoud, Z. Chen, and G. Bradshaw, "Departures from optimality: Understanding human analyst's information foraging in assisted requirements tracing," in *International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, May 2013, pp. 572–581.

[22] A. Mahmoud and N. Niu, "TraCter: A tool for candidate traceability link clustering," in *International Requirements Engineering Conference (RE)*, Trento, Italy, August-September 2011, pp. 335–336.

[23] S. Reddivari, Z. Chen, and N. Niu, "ReCVisu: A tool for clustering-based visual exploration of requirements," in *International Requirements Engineering Conference (RE)*, Chicago, IL, USA, September 2012, pp. 327–328.

[24] N. Niu, S. Reddivari, and Z. Chen, "Keeping requirements on track via visual analytics," in *International Requirements Engineering Conference (RE)*, Rio de Janeiro, Brazil, July 2013, pp. 205–214.

[25] A. Mahmoud and N. Niu, "Supporting requirements to code traceability through refactoring," *Requirements Engineering*, vol. 19, no. 3, pp. 309–329, September 2014.

[26] W. Wang, N. Niu, H. Liu, and Y. Wu, "Tagging in assisted tracing," in *International Symposium on Software and Systems Traceability (SST)*, Florence, Italy, May 2015, pp. 8–14.

[27] X. Jin, N. Niu, and M. Wagner, "Facilitating end-user developers by estimating time cost of foraging a webpage," in *International Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Raleigh, NC, USA, October 2017, pp. 31–35.

[28] N. Hussein, W. Wang, J. L. Nedelec, X. Wei, and N. Niu, "Unified profiling of attackers via domain modeling," in *International Workshop on Requirements Engineering for Investigating and Countering Crime (iRENIC)*, Beijing, China, September 2016, pp. 98–101.

[29] T. Bhowmik, N. Niu, W. Wang, J.-R. C. Cheng, L. Li, and X. Cao, "Optimal group size for software change tasks: a social information foraging perspective," *IEEE Transactions on Cybernetics*, vol. 46, no. 8, pp. 1784–1795, August 2016.

[30] D. Cepulis and N. Niu, "Creating socio-technical patches for information foraging: A requirements traceability case study," in *International Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Lisbon, Portugal, October 2018, pp. 17–21.

[31] A. Gupta, W. Wang, N. Niu, and J. Savolainen, "Answering the requirements traceability questions," in *International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, May 2018, pp. 444–445.

[32] N. Niu, W. Wang, A. Gupta, M. Assarandarban, L. D. Xu, J. Savolainen, and J.-R. C. Cheng, "Requirements socio-technical graphs for managing practitioners' traceability questions," *IEEE Transactions on Computational Social Systems*, vol. 5, no. 4, pp. 1152–1162, December 2018.

[33] W. Wang, N. Niu, M. Alenazi, J. Savolainen, Z. Niu, J.-R. C. Cheng, and L. D. Xu, "Complementarity in requirements tracing," *IEEE Transactions on Cybernetics*, vol. 50, no. 4, pp. 1395–1404, April 2020.

[34] W. Wang, A. Gupta, N. Niu, L. D. Xu, J.-R. C. Cheng, and Z. Niu, "Automatically tracing dependability requirements via term-based relevance feedback," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 342–349, January 2018.

[35] N. Niu, S. Brinkkemper, X. Franch, J. Partanen, and J. Savolainen, "Requirements engineering and continuous deployment," *IEEE Software*, vol. 35, no. 2, pp. 86–90, March/April 2018.

[36] M. Alenazi, D. Reddy, and N. Niu, "Assuring virtual PLC in the context of SysML models," in *International Conference on Software Reuse (ICSR)*, Madrid, Spain, May 2018, pp. 121–136.

[37] M. Alenazi, N. Niu, and J. Savolainen, "A novel approach to tracing safety requirements and state-based design models," in *International Conference on Software Engineering (ICSE)*, 2020.

[38] H. Gudaparthi, R. Johnson, H. Challa, and N. Niu, "Deep learning for smart sewer systems: Assessing nonfunctional requirements," in *International Conference on Software Engineering (ICSE)*, 2020.

[39] N. Niu, A. Koshoffer, L. Newman, C. Khatwani, C. Samarasinghe, and J. Savolainen, "Advancing repeated research in requirements engineering: a theoretical replication of viewpoint merging," in *International Requirements Engineering Conference (RE)*, Beijing, China, September 2016, pp. 186–195.

[40] C. Khatwani, X. Jin, N. Niu, A. Koshoffer, L. Newman, and J. Savolainen, "Advancing viewpoint merging in requirements engineering: A theoretical replication and explanatory study," *Requirements Engineering*, vol. 22, no. 3, pp. 317–338, September 2017.

[41] W. Wang, A. Gupta, and N. Niu, "Mining security requirements from common vulnerabilities and exposures for agile projects," in *International Workshop on Quality Requirements in Agile Projects (QuaRAP)*, Banff, Canada, August 2018, pp. 46–55.