A Bayesian Cognitive Approach to Quantifying Software Exploitability Based on Reachability Testing (Extended Version)

Guanhua Yan[1], Yunus Kucuk[1][2], Max Slocum[1], and David C. Last[3]

[1] Department of Computer Science, Binghamton University, State University of New York
{ghyan,ykucuk1,mslocum1}@binghamton.edu
[2] Defense Sciences Institute, Turkish Military Academy, Turkey
ykucuk@kho.edu.tr
[3] Resilient Synchronized Systems Branch, Air Force Research Laboratory, USA
david.last.1@us.af.mil

Abstract. Computer hackers or their malware surrogates constantly look for software vulnerabilities in the cyberspace to perform various online crimes, such as identity theft, cyber espionage, and denial of service attacks. It is thus crucial to assess accurately the likelihood that a software can be exploited before it is put into practical use. In this work, we propose a cognitive framework that uses Bayesian reasoning as its first principle to quantify software exploitability. Using the Bayes' rule, our framework combines in an organic manner the evaluator's prior beliefs with her empirical observations from software tests that check if the security-critical components of a software are reachable from its attack surface. We rigorously analyze this framework as a system of nonlinear equations, and henceforth perform extensive numerical simulations to gain insights into issues such as convergence of parameter estimation and the effects of the evaluator's cognitive characteristics.

1 Introduction

Software flaws are difficult, if not impossible, to avoid, either due to the limited cognitive capacities of the programmers to test all corner cases, or the fundamental weaknesses of the programming languages used. Software defects enable cybercriminals or their malware surrogates to perform a wide spectrum of malicious online activities, such as identity theft, cyber espionage, and denial of service attacks. As evidenced by numerous hacks that have occurred in the past, vulnerable software can result in significant economic losses and reputation damages. For instance, it was estimated that the revelation of the Shellshock vulnerability had led to one billion attacks [2], and an announced software vulnerability costs a firm an average loss of 0.5 percent value in stock price [33].

When a software system is put into practical use, its operator is concerned with the likelihood that it can be exploited maliciously. For security-critical applications, a software system can only be trusted for operational use if its operator's confidence level in its unexploitability exceeds a certain threshold, say, 99%. The challenge, then, is: *how can we derive such confidence levels to assist human operators with decision-making?* This problem is largely unexplored in the literature. There are some publicly available sources to find known software vulnerabilities, such as National Vulnerability Database [4], Exploit Database [5], and OSVDB (Open Sourced Vulnerability Database) [6]. However, these sources contain only known vulnerabilities in typically popular software, and thus cannot be solely relied upon to evaluate the security of a software system. Moreover, containing vulnerabilities does not necessarily mean that the software is exploitable in a certain running environment, as a successful software exploitation requires the existence of a realizable execution path from the attack surface of the program to its vulnerable software components [24, 27, 35].

Quantifiable measures of software exploitability can guide human operators in deciding whether it is sufficiently secure to put a software into operation. The Common Vulnerability Scoring System (CVSS) [28] is widely used in the industry, but its design is more of an *art* rather than a *science*. For example, it assesses the security of a vulnerable software with an overly simplistic equation: $BaseScore = 1.176 \times (3I/5 + 2E/5 - 3/2)$, where impact factor I and exploitability factor E take circumstance-specific values. Although this equation has surely been thoroughly meditated, there lack rigorous scientific arguments on why its parameters are so chosen.

In this work, we model the evaluation of software exploitability as a dynamic process done by an evaluator, who has her prior belief in software exploitability based upon some of its static features (e.g., its size, type, or some other metrics). Henceforth, she uses reachability testing tools to check whether there exists an injection vector from its attack surface that enables reachability of its security-critical components, such as a system call capable of privilege escalation or a potential buffer overflow vulnerability. The exploitability of the software is then characterized as the evaluator's subjective belief dynamically adjusted with the reachability testing results presented to her. During this process, the evaluator also continuously updates her perceptions about the performances of the tools used.

To model human cognition, we adopt a first-principled approach that integrates an evaluator's prior belief in software exploitability with her empirical observations from the reachability tests in a *Bayesian* manner. Bayesian reasoning is performed in a probabilistic paradigm, where given a hypothesis H and the evidence E, the posterior probability, or the probability of hypothesis H after seeing evidence E is calculated based upon the Bayes' rule: $\mathbb{P}\{H|E\} = \frac{\mathbb{P}\{E|H\} \cdot \mathbb{P}\{H\}}{\mathbb{P}\{E\}}$. Although there lacks evidence that humans reason in a Bayesian way at the neural level, psychological experiments show that humans behave consistently with the model at a functional level in a number of scenarios [17, 20, 29].

In a nutshell, our contributions can be summarized as follows:

- We propose a Bayesian cognitive framework that quantifies software exploitability as the evaluator's belief in whether an injection vector can be found from the attack surface of a software to enable the execution of a sensitive code block (e.g., one invoking a system call that leads to privilege escalation). The evaluator's belief is dynamically updated with the Bayes' rule, which uses the past performances of the reachability testing tools to calculate the likelihood functions for each hypothesis.

- We represent the Bayesian cognitive framework for quantifying software exploitability with a system of nonlinear equations, and rigorously analyze its time and space complexity, its sensitivity to the order of reachability tests, and the conditions under which the evaluator's belief in software exploitability improves or deteriorates.
- We use numerical simulations to analyze the Bayesian cognitive framework, including the convergence of the evaluator's beliefs, convergence of estimated parameters, effects of the evaluator's prior beliefs, effects of the ordering of software reachability tests, effects of dependency among different reachability testing tools, effects of short memory in parameter estimation, and effects of lazy evaluation. Our analysis shows that the nature of nonlinear equations leads to interesting observations that are not so intuitive.

From a high level, our work suggests a continuous and adaptive methodology for quantifiable cybersecurity, which is hard for an environment like the Internet that is open, dynamic and adversarial [34]. Although put in the context of software exploitability evaluation, the proposed Bayesian cognitive framework can be applied to various cybersecurity problems, such as malware detection and anomaly detection. Moreover, such cognitive frameworks allow us to further design autonomous systems that mimic the decision-making process of human defenders, thus preventing human errors.

2 Related Work

A large body of research has been dedicated to identifying security-sensitive software bugs in an efficient manner. One of the most widely used methods for finding software bugs in practice is black-box fuzzing, which generates malformed inputs in a brute-force manner to force crashes. The key challenge facing black-box fuzzing is lack of efficiency when dealing with large software systems, and there have been some recent works aimed at improving its performance [30, 16]. In contrast to black-box fuzzing, white-box fuzzing takes advantage of knowledge of the internal structures of the program to find software bugs. The key enabling technology behind effective white-box fuzzing is the so-called concolic execution or dynamic symbolic execution [13], which allows systematic exploration of program branches for whole-program security testing. Notable white-box fuzzing tools include EXE [12], KLEE [11] and SAGE [19, 18]. One step further, a few tools have been developed to automate the process of finding software exploits, such as APEG [10], AEG [8] and MAYHEM [15]. Many aforementioned tools can be used, directly or indirectly, for software reachability testing. Black-box fuzzing tools, for instance, can be used to test software reachability in an opportunistic manner. Symbolic or concolic execution tools can be adapted to find satisfiable paths reaching security-critical code blocks of interest.

Our work on quantifying software exploitability intersects with existing efforts on security metrics, which are valuable to strategic support, quality assurance, and tactical oversight in cyber security operations [22]. Although security metrics are important for cyber security to progress as a scientific field [25], it is hard to develop practically useful security metrics due to the dynamic and adversarial nature of the cyberspace [9, 22, 34]. As desirable properties of security metrics include objectivity and repeatability, software exploitability quantified by our proposed scheme does not qualify as a security metric. However, useful metrics indicative of software exploitability can be incorporated into our cognitive framework as the evaluator's prior belief. As the landscape of software exploitation is changing over time [26], these metrics may gradually lose their predictive power. Our cognitive framework allows the evaluator to adjust her beliefs with observations from new exploitation tests.

Our work finds inspirations from recent advances in modeling human cognition. A number of psychological experiments have shown that humans tend to behave consistently with the Bayesian cognitive model at the functional level [17, 20, 29]. Cognition-inspired methods have found a few applications in cyber security, such as malware family identification [23] and cyber-attack analysis [36]. Such cognition-based methods can be used in autonomous cyber defense systems to mimic the decision-making process of human operators and prevent human mistakes or their intrinsic cognitive biases [32].

3 Software Exploitation Based on Reachability Testing

An experienced hacker would narrow down the attack target to a few security-sensitive code blocks, a technique called *red pointing* [21]. Software exploitation can be characterized as the conjunction of both the existence of a bug and the possibility of the attacker hijacking the control flow [8]. With a software bug as the target, if there exists an execution path from the attack surface (which is controllable by the attacker) to invoke the software bug, the bug becomes exploitable. Consider the following C program with a buffer overflow bug:

```
#include <stdio.h>
#include <fcntl.h>
void innocent() { return; }
void vulnerable() { char buf[8]; gets(buf); }
int main(int argc, char** argv) {
     if (argc != 2) { return -1; }
     int fn = open(argv[1], O_RDONLY);
     char c, d = 0;
     int i;
     for (i = 0; i < 10; i++) {
         if(read(fn, &c, 1) == 1) d = d^c; else break;
     }
     if (d == 0) vulnerable(); else innocent();
     close(fn);
}
```

To reach `vulnerable()` with a buffer overflow bug, we need to find an input file the XOR of whose first 10 bytes is 0. We tried the following on a commodity PC:

Black-box fuzzing: A black-box fuzzer randomly generates input files to force program crashes. We add `assert(0);` at the beginning of function `vulnerable()` to cause a crash when it is called, and then use BFF [14] to fuzz against the program. Using a single seed file of size 1,805 bytes, BFF can find the first crash within a second.

Symbolic execution: Symbolic execution does not need to execute the program concretely. Rather, it relies upon symbolic evaluation to find an input that causes a part of the program to be executed. We use the Z3 tool developed by Microsoft Research [31] to find a satisfiable condition that enables the execution of function `vulnerable()`. As Z3 does not support the `char` type explicitly, we use bit-vectors of size 8 (in Z3 parlance, they are defined with: `Z3_sort bv_sort = Z3_mk_bv_sort(ctx, 8)` where `ctx` is a Z3 context) to perform bit-wise XOR operations. With 10 symbolic variables of type *bv_sort* defined, Z3 can find within a few milliseconds their assignments such that the condition for entering function `vulnerable()` is satisfied.

Concolic execution: Concolic execution combines symbolic execution with concrete execution to speed up code exploration. We first try the CREST tool [1] to find solutions to the 10 symbolic variables of type `CREST_char`, each corresponding to a byte read from a file. However, as CREST uses Yices 1 as its SMT solver for satisfiability of formulas [7], which does not support bit-vector operations, it does not find a condition that leads to the execution of function `vulnerable()`. Another popular concolic execution tool is KLEE [11], which works on object files in the LLVM bitcode format and uses the STP solver supporting bit-vectors and arrays [3]. Similarly, by defining 10 symbolic variables using `klee_make_symbolic`, each corresponding to a byte read from the input file, we are able to use KLEE to find quickly their proper assignments that enable the execution of function `vulnerable()`.

For a large and complex software, some of the tools may not find exploits enabling reachability of its security-critical components. A security evaluator may need multiple tools for a software exploitation task, and intuitively, her memory of the past performances of these tools affects her evaluation of software exploitability.

4 A Bayesian Cognitive Framework

Motivated by the example in the previous section, we model software exploitation as a process of finding a proper injection vector in the attack surface of a software that enables its execution to reach one of its security-critical code blocks, using some reachability testing tools. Our goal is to quantify software exploitability as the likelihood that, given a security-critical target in the software, there exists such an injection vector that successfully leads to its execution. We assume that the evaluation of software exploitability is performed by an evaluator. Intuitively, if she has already found such an injection vector, her perception of the exploitability of this software is certain. Otherwise, she is *uncertain* about the exploitability of the software: there may exist an execution path that reaches the target from the attack surface but she just cannot find it at the moment. The evaluator may proceed to use some other tools to check the existence of such an injection vector, and with more failed attempts, she should be increasingly confident in the notion that the security-critical target of the software system is not exploitable.

Some notations are needed to describe the probabilistic model characterizing this cognitive process. We define the software-target pair (s, x) as an *exploitation task*, whose goal is to find whether target x is reachable in software s from its attack surface. We consider a null hypothesis $H_0(s, x)$, which simply states that target x is *unreachable* in software s from its attack surface. Hence, the unexploitability of target x in software s is quantified by the probability with which the null hypothesis is true, i.e., $\mathbb{P}\{H_0(s, x) \text{ is true}\}$, or simply $\mathbb{P}\{H_0(s, x)\}$. For ease of presentation, we let the null hypothesis $H_0(s, x)$ be the evaluator's *belief* in the unreachability of target x in software s and $\mathbb{P}\{H_0(s, x)\}$ her *belief level*.

Available to the evaluator is also a list of reachability testing tools, as discussed in Section 3, which can be used to find an injection vector from a software's attack surface to reach a given security-critical target in a software. Let Z denote such a list of tools, each of which works as follows: given target x in software s, tool $z \in Z$ either outputs that x is not reachable in s from its attack surface, or an injection vector that it detects to be able to reach target x. Given an injection vector v by a tool, the evaluator can execute the software with the injection vector v to validate whether target x can be reached. Like any other security detector, a reachability testing tool may wrongly report that target x is unreachable in software s, or misdetect a wrong injection vector as being able to reach target x.

Table 1. Tool parameters (IV: injection vector)

Truth / Result	Unreachable	Reachable, correct IV	Reachable, wrong IV
Unreachable	α	0	$1 - \alpha$
Reachable	β	γ	$1 - \beta - \gamma$

We thus model the performance of a reachability testing tool as probabilities in Table 1. For each reachability testing tool, three parameters, α, β, and γ, are used to model its performance: (1) *The truth is that the target is unreachable*. A binomial process is used to characterize the output of the tool, which returns a result of being unreachable with probability α, and a result of being reachable with probability $1 - \alpha$. (2) *The truth is that the target is reachable*. The tool behaves as a multinomial process: it classifies the target as being unreachable with probability β, as being reachable with a correct injection vector with probability γ, and as being reachable with a wrong injection vector with probability $1 - \beta - \gamma$.

The rationale behind choosing the binomial and multinomial processes in our model is two-fold: they not only lead to a parsimonious model of human recognition of tool performances (by simple counting), but also provide algebraic convenience as their conjugate priors are well known. In a more fine-grained model, for the same tool z, the evaluator may associate different parameter values with some properties of the software (e.g., its size, type, or some other metrics). To deal with such subtleties, for each tool z, the evaluator can associate different values of parameters $\alpha^{(z,k)}$, $\beta^{(z,k)}$, and $\gamma^{(z,k)}$ when it is applied on software of type k. Moreover, to reflect the dynamics of these parameters, we use subscript t to indicate their values at time t. For example, $\alpha_t^{(z,k[s])}$ gives the value of parameter α at time t when tool z is used on the type of software $k[s]$.

Next we discuss how the evaluator, after using tool z for a new reachability test, updates her posterior belief in the reachability of target x in software s. Let the new observation made at time t be O_t, which falls into one of the following types:

- **Type** E_0: The tool detects target x to be unreachable in software s.
- **Type** E_1: The tool detects that target x is reachable in software s, and also returns an injection vector v, which is verified to be *true* by the evaluator.
- **Type** E_2: The tool detects that the target x is reachable in software s, and also returns an injection vector v, which is verified to be *false* by the evaluator.

After performing a reachability test with tool z and observing O_t from the test at time t, her belief level in the unreachability of target x in software s is updated to be the posterior probability $\mathbb{P}\{H_0(s,x)|O_t\}$ according to Eqs. (1-3) in Figure 1.

$$\mathbb{P}\{H_0(s,x)|O_t = E_0\} = \frac{\mathbb{P}\{H_0(s,x)\} \cdot \alpha_t^{(z,k[s])}}{\mathbb{P}\{H_0(s,x)\} \cdot \alpha_t^{(z,k[s])} + (1 - \mathbb{P}\{H_0(s,x)\}) \cdot \beta_t^{(z,k[s])}} \tag{1}$$

$$\mathbb{P}\{H_0(s,x)|O_t = E_1\} = 0 \tag{2}$$

$$\mathbb{P}\{H_0(s,x)|O_t = E_2\} = \frac{\mathbb{P}\{H_0(s,x)\} \cdot (1 - \alpha_t^{(z,k[s])})}{\mathbb{P}\{H_0(s,x)\} \cdot (1 - \alpha_t^{(z,k[s])}) + (1 - \mathbb{P}\{H_0(s,x)\}) \cdot (1 - \beta_t^{(z,k[s])} - \gamma_t^{(z,k[s])})} \tag{3}$$

Fig. 1. Calculation of posterior probability after seeing the result from a reachability test

The calculation of Eqs. (1-3) is based upon the Bayes' rule and the performance of the reachability testing tool in Table 1. In Eq. (1), the observation is that the tool detects the target to be unreachable. As the hypothesis $H_0(s,x)$ states that the target is unreachable, the probability that the observation results from the hypothesis being true is $\alpha_t^{(z,k[s])}$. If the opposite hypothesis holds (the target is reachable), the observation occurs with probability $\beta_t^{(z,k[s])}$. Hence, Eq. (1) naturally follows based on the Bayes' rule. Moreover, when it is observed that the tool classifies the target to be reachable with a correct injection vector, it is certain that hypothesis $H_0(s,x)$ must not hold any more. This can be confirmed by Eq. (2) as $\mathbb{P}\{E_1|H_0(s,x)\}$ equals 0. Similarly, we can reason about the case when the tool classifies the target as being reachable but provides a wrong injection vector, and derive Eq. (3).

5 Parameter updating

In this section, we discuss how the evaluator dynamically updates the values of the performance parameters (i.e., α, β, and γ) associated with each reachability testing tool based on the Bayes' rule. To evaluate the performance of a reachability testing tool, it would help if the ground truth is known to the evaluator. For example, if it is known that target x is surely reachable from the attack surface of software s, any tool that reports it being unreachable has a false negative error. One important observation, however, is that *if it is true that target x is unreachable in software s, it may never be verifiable by the evaluator for a large software, although the opposite is not true: as long as a single injection vector is found to reach target x, it is certain that the target must be reachable.* Hence, when no verifiable injection vector has been found yet to reach target x from the attack surface of software s, a *"relative fact"* reflecting whether a target has been found reachable is used to replace the truth in Table 1. Therefore, for each reachability testing tool $z \in Z$ used on software of type k, the evaluator keeps a *performance counting table*, or $PCT^{(z,k)}$, which contains five performance counters $c_0^{(z,k)}$, ..., $c_4^{(z,k)}$ as in Table 2. When the context is clear, we drop the superscript (z,k).

The evaluator performs a sequence of software reachability tests, $Q = \{q_0, q_1, ..., q_t, ...\}$, where in $q_t = (s_t, x_t, z_t, o_t)$, tool z_t is used to test the reachability of x_t in software

Table 2. The performance counting table for tool z used on software of type k, i.e., $PCT^{(z,k)}$.

"Relative fact" / Result	Unreachable	Reachable, correct IV	Reachable, wrong IV
Unreachable	$c_0^{(z,k)}$	N/A	$c_1^{(z,k)}$
Reachable	$c_2^{(z,k)}$	$c_3^{(z,k)}$	$c_4^{(z,k)}$

s_t at time step t with observed test result o_t. For ease of explanation, we further define subsequences of software exploitation tests, each corresponding to a specific software exploitation task (s, x):

$$Q_{s,x} = \{q_t \mid s_t = s \wedge x_t = x\}, \tag{4}$$

and the first element in $Q_{s,x}$ is given as $Q_{s,x}[0]$.

For exploitation task (s, x), parameters are updated based upon its mode $m(s, x)$: *pre-exploitation* and *post-exploitation*. In the pre-exploitation mode, the evaluator has not found any injection vector that enables reachability of target x in software s, and by contrast, in the post-exploitation mode, such an injection vector has already been found. Initially, for every software exploitation task (s, x) its mode $m(s, x)$ is set to be *pre-exploitation*.

Consider the software reachability tests in Q sequentially. Given a new test (s, x, z, o) in Q, which corresponds to the i-th one in $Q_{s,x}$ (i.e., $Q_{s,x}[i] = (s, x, z, o)$), the evaluator uses the following rules to update the performance counters in table $PCT^{(z,k[s])}$, where $k[s]$ is the type of software s:

- **Rule I** applies to the case when $o = E_0$. If $m(s, x)$ is *pre-exploitation*, $c_0^{(z,k[s])}$ increases by 1; otherwise, $c_2^{(z,k[s])}$ increases by 1.
- **Rule II** applies to the case when $o = E_1$. If $m(s, x)$ is *post-exploitation*, $c_3^{(z,k[s])}$ increases by 1. Otherwise, if $m(s, x)$ is *pre-exploitation*, the evaluator has just found an injection vector to reach target x in software s. After increasing $c_3^{(z,k[s])}$ by 1, mode $m(s, x)$ is changed from *pre-exploitation* to *post-exploitation*. *During this change of mode, the evaluator also needs to update the performance counters for those tools that have been previously used to test the software, as the "relative fact" that has been used to update these counters previously turns out to be false.* Hence, for every j with $0 \leq j < i$, supposing that $Q_{s,x}[j] = (s, x, z', o')$, the following *revision steps* are applied: (1) if $o' = E_0$, then decrease $c_0^{(z',k[s])}$ by 1 and increase $c_2^{(z',k[s])}$ by 1; (2) if $o' = E_2$, then decrease $c_1^{(z',k[s])}$ by 1 and increase $c_4^{(z',k[s])}$ by 1. Note that it is impossible to have $o' = E_1$ (otherwise, the mode must have already been changed to *post-exploitation* after o' is seen). Hence, the evaluator needs to revise the performance counts based on the newly found truth that target x is reachable from the attack surface of software s.
- **Rule III** applies to the case when $o = E_2$. If $m(s, x)$ is *pre-exploitation*, $c_1^{(z,k[s])}$ increases by 1; otherwise, $c_4^{(z,k[s])}$ increases by 1.

The performance counters in table $PCT^{(z,k)}$ can be used to estimate the parameters $\alpha_t^{(z,k)}$, $\beta_t^{(z,k)}$, and $\gamma_t^{(z,k)}$ at the current time t. We let the values of the performance counters in table $PCT^{(z,k)}$ at time t be $c_i^{(z,k)}(t)$, for $i = 0, ..., 4$. Using a frequentist's view, parameters $\alpha_t^{(z,k)}$, $\beta_t^{(z,k)}$, and $\gamma_t^{(z,k)}$ could be estimated as their relative frequencies. When few tests have been done, however, the estimated values of $\alpha_t^{(z,k)}$, $\beta_t^{(z,k)}$, and $\gamma_t^{(z,k)}$ as derived may not be sufficiently reliable to characterize the performance of the reachability testing tool. This resembles the scenario that a person, whose prior belief is that any coin is fair, would not believe that the coin will always produce `head` even after seeing three `heads` in a row.

Our model, again, takes the evaluator's prior belief into account when estimating these parameters. After tool z is used to test whether target x is reachable in software s, which is of type k, if $m(s, x)$ is still *pre-exploitation*, the truth may not be known to the evaluator. Without knowing the truth, the evaluator relies on the "relative fact" that target x is not reachable from the attack surface of software s. Therefore, depending on the current mode of exploitation task (s, x), she updates the parameters as follows:

- If $m(s, x)$ is *pre-exploitation*, tool z works as a Binomial process where it returns a result of being unreachable with probability $\alpha^{(z,k)}$. As the conjugate prior for a Binomial process is a Beta distribution, we assume that the prior for parameter $\alpha^{(z,k)}$ takes a $Beta(d_0^{(z,k)}+1, d_1^{(z,k)}+1)$ distribution. We use the MAP (Maximum A Posteriori) estimate to update $\alpha^{(z,k)}$:

$$\alpha_t^{(z,k)} = \frac{d_0^{(z,k)}+c_0^{(z,k)}(t)}{d_0^{(z,k)}+c_0^{(z,k)}(t)+d_1^{(z,k)}+c_1^{(z,k)}(t)} \tag{5}$$

- If $m(s, x)$ is *post-exploitation*, tool z behaves as a multinomial process where it returns being unreachable with probability $\beta^{(z,k)}$, being reachable with a correct injection vector $\gamma^{(z,k)}$, and being reachable with a wrong injection vector $1 - \beta^{(z,k)} - \gamma^{(z,k)}$. Similarly, as the conjugate prior for a multinomial process is the Dirichlet distribution, we assume that the prior for parameter $(\beta^{(z,k)}, \gamma^{(z,k)})$ follows a Dirichlet distribution $Dir(d_2^{(z,k)}+1, d_3^{(z,k)}+1, d_4^{(z,k)}+1)$. We again use the MAP estimate to update $\beta^{(z,k)}$ and $\gamma^{(z,k)}$:

$$\beta_t^{(z,k)} = \frac{d_2^{(z,k)}+c_2^{(z,k)}(t)}{\sum_{i=2}^{4} d_i^{(z,k)}+\sum_{i=2}^{4} c_i^{(z,k)}(t)} \tag{6}$$

$$\gamma_t^{(z,k)} = \frac{d_3^{(z,k)}+c_3^{(z,k)}(t)}{\sum_{i=2}^{4} d_i^{(z,k)}+\sum_{i=2}^{4} c_i^{(z,k)}(t)} \tag{7}$$

The evaluator assumes target x to be unreachable from the attack surface of software s if mode $m(s, x)$ is pre-exploitation, and this assumption is used as the relative fact to update the performance counters in related PCTs. However, when a later test finds an exploitation for the task (s, x), which invalidates the assumption, the parameters of those tools whose values have been previously estimated based upon this relative fact should be updated to reflect this change of mode. Mechanically, however, the evaluator can simply maintain PCTs like Table 2, and whenever it is necessary to use parameters α, β, and γ in Equations (1-3), the tables are used to calculate their latest values based on Equations (5-7).

6 Model Analysis

Space complexity. The space used in the cognitive model includes those PCTs that the evaluator uses to keep the aggregate results from previous software reachability tests.

It is noted that the prior information for parameter updating (i.e., d_0-d_4) can be put in the tables as initial values; hence, each entry in the table represents $c_i^{(z,k)}(t) + d_i^{(z,k)}$ where $0 \leq i \leq 4$. Supposing that there are $|Z|$ reachability testing tools and $|K|$ software types, as each PCT contains 5 entries (see Table 2), it requires $5|Z||K|$ to store the tables. Clearly, as the space is linear with $|K|$, more fine-grained categorization of software would bring more cognitive burden to the evaluator unless auxiliary methods are used to help remember these tables.

For every exploitation task (s, x), it is necessary to remember the evaluator's belief level $\mathbb{P}\{H_0(s, x)\}$ and its current mode $m(s, x)$. When an exploitation task is in the *pre-exploitation* mode, the evaluator also needs to remember the tools that have been previously used for the task, so if later an exploit is found, the evaluator can take the revision steps to correct the performance counters associated with these tools (Rule II of parameter updating). Therefore, if no specific ordering scheme on the exploitation tools is used, the amount of tests that the evaluator has to remember may be large, and in the worse case, it is $|Q|$.

To alleviate her cognitive burden, the evaluator may use auxiliary devices (e.g., papers) for remembering the information needed in the model, or simplify the model. For example, all the tools are numbered, and for every exploitation task, these tools are always used in an increasing order. Rules can be used to check if a tool is applicable for an exploitation task. Hence, when the mode of an exploitation task changes from *pre-exploitation* to *post-exploitation*, the evaluator can simply revise the PCTs of those applicable tools that are numbered lower than the one finding the exploitation.

Time complexity. Given the input Q, it is assumed that executing each of Equations (1-3) takes a constant amount of time. For an exploitation task (s, x), changing its mode from *pre-exploitation* to *post-exploitation* requires updating the performance counters of those tools that have previously been used on them. However, for each reachability test in Q, revision of its result occurs at most once. Therefore, the time complexity of the model is $O(|Q|)$.

We can thus establish the following theorem regarding the complexity of the model:

Theorem 1. *The space and time complexity of the cognitive model is $O(|Z||K| + n + |Q|)$ and $O(|Q|)$, respectively, where $|Z|$ is the number of reachability testing tools, $|K|$ is the number of software types, n is the number of exploitation tasks, and $|Q|$ is the total number of reachability tests done by the evaluator.*

6.1 Order Sensitivity

Equations (1-3) and (5-7) form a complex nonlinear system, whose input is comprised of sequence Q, the initial states of the PCTs for all tools in Z, and the prior values of $\mathbb{P}\{s, x\}$ for every exploitation task (s, x). We say that the cognitive model is *order insensitive* if no matter how we change the order of tests in Q, the following conditions are satisfied after all tests: (1) the evaluator's final belief level for every exploitation task is the same, and (2) the states of all the PCTs are the same. It is noted that the mode of each exploitation task must not change with the order of tests in Q: For any exploitation task (s, x), if its mode is *post-exploitation* before tests in Q, its mode remains the same after all tests in Q; otherwise, if there exists any test in Q for this task that leads to observation E_1, regardless of its order in Q, the mode of the task must be changed to *post-exploitation*, or otherwise if no such test exists, its mode should be *pre-exploitation*.

To understand under what circumstances the cognitive model is order insensitive, we first start with a simple case where there are only two reachability tests in Q. We can establish the following lemma (proof in Appendix A):

Lemma 1. *For any* $Q = [(s_0, x_0, z_0, o_0), (s_1, x_1, z_1, o_1)]$ *and* $Q' = [(s_1, x_1, z_1, o_1), (s_0, x_0, z_0, o_0)]$*, if* $(s_0, x_0) = (s_1, x_1)$ or $z_0 \neq z_1$*, the cognitive model is order insensitive.*

Now we consider the general case of array Q which may have more than two tests. According to Lemma 1, for any two consecutive reachability tests in a sequence, as long as they do not use the same reachability testing tool on two different exploitation tasks, we can swap their order. We call such a swapping of consecutive reachability tests a *safe swapping*. Given a sequence of reachability tests in Q, we can freely perform safe swappings on two consecutive tests without affecting the evaluator's final beliefs. We can thus establish the following theorem (proof in Appendix B):

Theorem 2. *For any sequence* Q *of software exploitation tests and* Q' *one of its permutations, assume that for every reachability testing tool, the relative order of reachability tests using this tool is the same in* Q *and* Q'*. Then the evaluator's final belief in every exploitation task must be the same after finishing* Q *and* Q'*.*

6.2 Exploitability Analysis

We now consider under what conditions a new reachability test, (s, x, z, o), improves the posterior probability $\mathbb{P}\{H_0(s, x)\}$. We consider the following cases. Without loss of generality, we drop the subscripts of the parameters.

Observation $o = E_0$: Given Eq. (1), in order to have $\mathbb{P}\{H_0(s, x) \mid O_t = E_0\} > \mathbb{P}\{H_0(s, x)\}$, we must have both $\alpha > \beta$ and $0 < \mathbb{P}\{H_0(s, x)\} < 1$. If $\mathbb{P}\{H_0(s, x)\} = 1$, the evaluator is certain that the target is not reachable a priori and thus any new evidence does not improve the posterior probability. On the other hand, if $\mathbb{P}\{H_0(s, x)\} = 0$, the Bayes' rule tells us that the posterior probability is also 0. With $\alpha > \beta$, it means that an unreachable target is detected to be unreachable with a higher probability than a reachable target being mistakenly classified as unreachable. Therefore, when a new test shows that the target is unreachable, it is better to use the former as the explanation than the latter, which suggests that the posterior probability $\mathbb{P}\{H_0(s, x) | E_0\}$ becomes higher after the test.

Observation $o = E_1$: Given Eq. (2), if the mode is still *pre-exploitation*, then seeing the test result lowers the evaluator's belief; otherwise, her belief level remains to be 0.

Observation $o = E_2$: Given Eq. (1), in order to have $\mathbb{P}\{H_0(s, x) \mid O_t = E_2\} > \mathbb{P}\{H_0(s, x)\}$, we must have: $\alpha < \beta + \gamma$ and $0 < \mathbb{P}\{H_0(s, x)\} < 1$. The same argument holds when $\mathbb{P}\{H_0(s, x)\}$ = 0 or 1 as in the case when $o = E_0$. With $\alpha < \beta + \gamma$ or equivalently $1 - \alpha > 1 - (\beta + \gamma)$, it is more likely that an unreachable target is detected by the tool to be reachable with a wrong injection vector than a reachable target being detected as reachable but with a wrong input vector; hence, given the same observation E_2, it is better to use the former than the latter to explain the observation.

The above analysis leads to the following theorem:

Table 3. Parameter settings in baseline cases

Parameter	Value	Parameter	Value
Number of tools	100	Initial counts in PCTs	All 1's
Number of software	100	Parameter α	[0.2, 0.4, 0.6, 0.8]
Prior belief level	0.5	Parameters β, γ	[0.1, 0.2, 0.3, 0.4, 0.5]
Parameter ϕ	0.3	Test ordering	Order by software then tools

Theorem 3. *For an exploitation task in a pre-exploitation mode, with a reachability testing tool of parameters α, β, and γ for the type of software in the task, the test result by this tool boosts the evaluator's belief level if and only if the evaluator's prior belief is in $(0, 1)$ and we have $\alpha > \beta$ if E_0 is observed or $\alpha < \beta + \gamma$ if E_2 is observed.*

7 Numerical Results

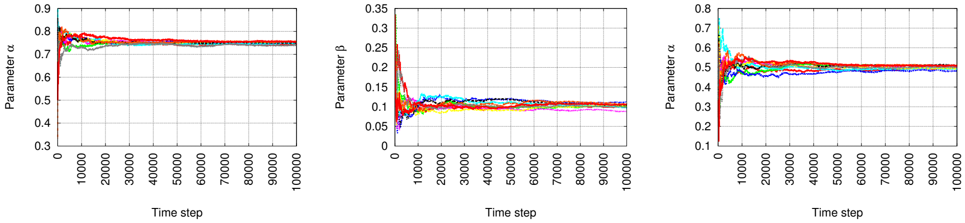
We perform experiments that simulate the Bayesian cognitive model, a system of nonlinear equations. The baseline configuration of an experiment is shown in Table 3. The reachability testing tools are those discussed in Section 3. As a reachability testing software may behave differently under different configurations, they are treated as different tools in our experiments. Parameter ϕ denotes the true probability that an exploitation task is achievable. For the test ordering, the tests are first ordered by the software to be exploited and then for each software, it is tested with the tools in the same order. The experiments mentioned in this section use parameter settings in Table 3 unless stated otherwise. We assume that the tests performed by all the tools are independent. For each tool, as the initial counts in its PCTs are all 1's, the evaluator's prior estimations of α, β, and γ are 1/2, 1/3, and 1/3, respectively.

Convergence of estimated parameters α, β, γ. In this set of experiments, we study how the estimated parameters converge over time. We consider 10 reachability testing tools, which are used to test 10,000 software. For each tool, its parameters α, β, and γ have true values, 0.75, 0.1, and 0.5, respectively. The others are the same as in Table 3.

Figure 2 shows the convergence of the parameters estimated by the evaluator. *We observe that the estimation of each parameter eventually converges towards its true value, but the convergence occurs slowly.* For instance, even after performing reachability tests for 1000 software (i.e., after time step 10000 as each software uses 10 time steps, one by each tool), the estimated value of each parameter is still not very stable. Also, although the 10 tools have the same true values for their parameters, there is significant variation among these tools after 10000 reachability tests.

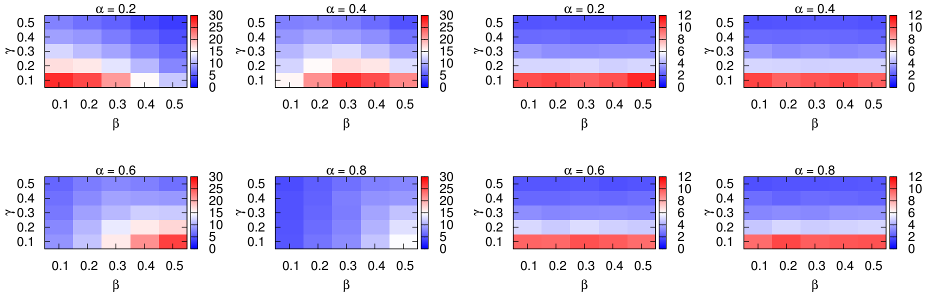
Convergence of belief levels. In this set of experiments, we study the convergence of the evaluator's belief levels. Figure 3 presents, for each combination of parameter settings, the average number of tests the evaluator needs to reach a belief level of 99% for a truly unexploitable software (left), along with the average number of tests to find an exploit for a truly exploitable software (right).

We first examine the results for truly unexploitable software. From Figure 3(1), we observe that for a truly unexploitable software, the average number of tests required to reach a belief level of 99% ranges from 3.6 to 27.2, showing a wide variation across different combinations of parameter settings. We also observe that given the same parameters α and β, increasing γ reduces the number of tests needed. This is because



(1) Parameter α (2) Parameter β (3) Parameter γ

Fig. 2. Convergence of parameters α, β, and γ. The true values of these parameters are 0.75, 0.1, and 0.5, respectively. In each time step, a reachability test is performed. The ranges of these estimated parameters among the 10 tools in the last time step are 0.0152, 0.0248, and 0.0292, which are 2.0%, 24.8%, and 5.8% of their true values, respectively.



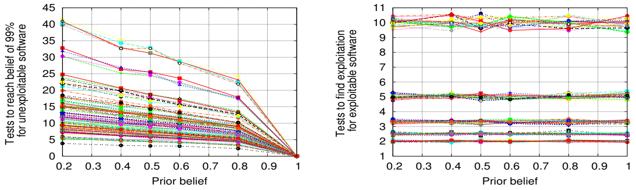
(1) Left case: unexploitable software (2) Right case: exploitable software

Fig. 3. Convergence of belief levels. The left case gives the average number of tests before the evaluator's belief level reaches 99% for a truly unexplotable software, and the right one the average number of tests before the evaluator's belief level reaches 0% for a truly exploitable software.

that the evaluator's belief level is affected by γ only through Eq. (3), where a higher γ boosts her belief level. The observation also agrees well with Theorem 3,

The effects of parameter β, however, are not as straightforward with the same α and γ. We observe that when α is small, a higher β reduces the number of tests required, but when α is large, increasing β would also increase the number of tests. This can be explained as follows. Note that both observations E_0 and E_2 allow β to affect the evaluator's belief. When α is higher, the number of observations of type E_0 increases, and the importance of Eq. (1) becomes higher, where a higher β decreases the evaluator's belief level; by contrast, when α is smaller, the number of observations of type E_2 increases, which increases the importance of Eq. (3), where a higher β increases the evaluator's belief level.

We next study the results for truly exploitable software. From Figure 3(2), we observe that the range of tests required for the subject to find a successful exploit is from 1.9 to 11.0. The dominating factor is γ, where a higher γ reduces the number of tests needed. This agrees well with our intuition that with tools that are more capable of finding exploits, the evaluator needs fewer tests to find exploits.



(1) Unexploitable software (2) Exploitable software

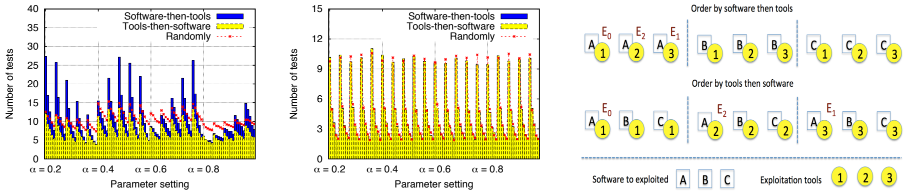
Fig. 4. Effects of prior beliefs

Effects of prior beliefs. In this set of experiments, we vary the evaluator's prior belief levels to study their effects. Figure 4 presents the average number of tests for the evaluator to reach a belief level of 99% for a truly unexploitable software and the average number of tests to find an exploit for a truly exploitable software. For the former, it is observed that a higher prior belief reduces the number of tests to reach a certain belief level. This is because regardless of the observation types (E_0 or E_2), the posterior belief increases monotonically with the prior belief as seen in both Eqs. (1) and (3). At one extreme, if the evaluator holds her prior belief firmly that the target must be reachable, any observation that no exploitation has been found against the software does not change that belief at all. That is to say, the number of tests for her to reach a belief of 99% would be infinity. At the other extreme, if the evaluator is certain that the software is not exploitable, obviously it does not need any test for her to reach a belief level of at least 99%.

Furthermore, as reachability tests are performed independently, the average number of tests to find an exploit for a truly exploitable software is always $1/\gamma$, irrespective of the evalutor's prior belief level. This is confirmed by Figure 4(2), where the evaluator's belief level does not change with the average number of tests needed to find an exploit.

Effects of test ordering on belief convergence. We now study how changing the order of software reachability tests affects the evaluator's belief convergence. We perform three groups of experiments: In the first group (*order-by-software-then-tools*), the tests are first ordered by the software to be exploited and then for each software, we test it using 100 tools in the same order. In the second group (*order-by-tools-then-software*), the tools are first ordered, and then for each tool, it is used to exploit the 100 software consecutively in the same order. In the third group *order-randomly*, the reachability tests are ordered randomly. Figure 5 again shows the average number of tests needed to reach a belief level of 99% for a truly unexploitable software (left) and the average number of tests to find an exploit for a truly exploitable software (right).

Interestingly, we observe that given a truly unexploitable software, on average it takes more tests to reach a certain belief level in the group of order-by-software-then-tools than those in the group of order-by-tools-then-software. The key difference is illustrated by a simple example shown in Figure 5(3), where three tools, 1, 2, and 3, are used to test software A, B, and C. The test results of applying tools 1, 2, and 3 on software A are E_0, E_2, and E_1, respectively. For ease of explanation, we assume that before the tests, the performance counters c_0, c_1, c_2, c_3, and c_4 of all the tools are all initialized to be 1. If the tests are first ordered by software and then tools (the upper row), then the first three tests are performed with the three tools on software A. After



(1) Unexploitable software (2) Exploitable software (3) Illustration of effects of ordering

Fig. 5. Effects of test ordering on belief convergence. The left case gives the average number of tests needed to reach a belief level of 99% for a truly unexploitable software, and the right one the average number of tests to find an exploit for a truly exploitable software. For each α setting, the tests are sorted by the increasing order of tuple (β, γ).

these three tests, because tool 3 finds an exploitable path, the performance counters of the three tools are: (1,1,2,1,1), (1,1,1,1,2), and (1,1,1,2,1). These counts will be used to update the posterior belief levels of software B and C later. By contrast, if the tests are first ordered by tools and then software (the bottom row), after the first test (tool 1 used on software A), the performance counters of tool 1 becomes (2,1,1,1,1) and these counts are used to update the posterior belief levels on software B and C in the second and third tests. Similarly, after the fourth test (tool 2 used on software A), the performance counters of tool 2 becomes (1,2,1,1,1), which will be used to update the posterior beliefs on software B and C next.

Hence, when the tests are first ordered by software and then tools, if any tool can find an exploitable path of software, this fact can change the mode of the software from *pre-exploitation* to *post-exploitation* and the performance counters of the tools previously used to test this software are updated to reflect this fact before they are used for testing other software. In contrast, if the tests are first ordered by the tools and then software, when the mode of the software is changed from *pre-exploitation* to *post-exploitation*, the performance counters of the tools previously used to test this software were updated assuming that the software is unexploitable, and then used to update the posterior beliefs of those software that were tested with these tools before the mode change.

How does such a difference affect the evaluator's posterior belief levels? For the same observation E_0, the performance counter c_0 increases by 1 if the mode is *pre-exploitation*, or c_2 increases 1 if the mode is *post-exploitation*. As we have:

$$\frac{\beta}{\alpha} = \frac{c_2/(c_2+c_3+c_4)}{c_0/(c_0+c_1)} = \frac{1+c_1/c_0}{1+(c_3+c_4)/c_2}, \tag{8}$$

post-exploitation updating increases $\frac{\beta}{\alpha}$ compared to pre-exploitation updating, which further decreases the evaluator's belief level after she sees E_0 according to Eq. (1).

Similarly, for the same observation E_2, the performance counter c_1 increases by 1 in the mode of *pre-exploitation*, or c_4 increases 1 in the mode of *post-exploitation*. Since

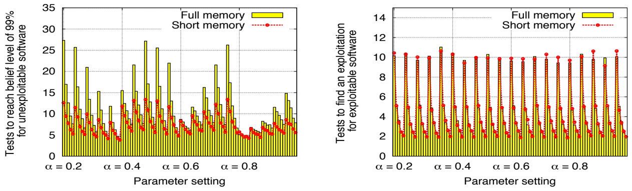
$$\frac{1-\beta-\gamma}{1-\alpha} = \frac{c_4/(c_2+c_3+c_4)}{c_1/(c_0+c_1)} = \frac{1+c_0/c_1}{1+(c_2+c_3)/c_4}, \tag{9}$$

post-exploitation updating increases $\frac{1-\beta-\gamma}{1-\alpha}$ compared to pre-exploitation updating, which further decreases the evaluator's belief level after E_2 is observed according to Eq. (3).

In summary, post-exploitation updating always reduces the evaluator's belief level for the software exploitation task at hand. This explains why more tests are needed for the evaluator to reach a certain belief level when tests are first ordered by software and then tools than when they are first ordered by the tools and then software, because the former case has more post-exploitation updatings than the latter, as seen in Figure 5(1). To confirm this, we did the experiments without any observations of type E_1 and then the differences in Figure 5(1) between order-by-software-then-tools and order-by-tools-then-software disappeared. Hence, there seems to be an irony: *postponing knowing that some software are exploitable helps improve the evaluator's belief level in the unexploitability of the others!*

In Figure 5(2), we present the average number of tests for the evaluator to find a successful exploit for a truly exploitable software. It is seen that the effect of the order of the reachability tests is little. This is because the test results by different reachability testing tools are assumed to be independent. With a probability of γ for any tool to find the proper injection vector for an exploitable software, the average number of tests needed is thus $1/\gamma$.

Effects of short memory. Recall that in the basic cognitive model, the evaluator has to remember the test results for each software exploitation task that is still in the pre-exploitation mode. According to Theorem 1, this may cause high cognitive burden to the evaluator. Hence, in a new set of experiments, we study the effects of short memory, with which the evaluator omits the revision steps in Rule II of parameter updating.

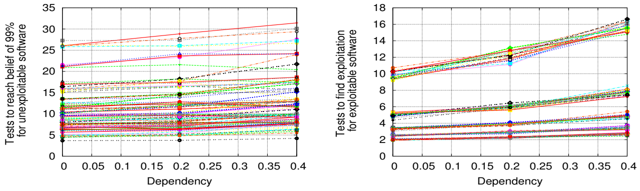


(1) Left case (2) Right case

Fig. 6. Effects of short memory. The left case shows the average number of tests needed to reach a belief level of 99% for an unexploitable software, and the right case the average number of tests needed to find an exploit for an exploitable software.

Figure 6 shows the effects of having a short memory in parameter updating on the evaluator's belief convergence. We observe that *due to a shorter memory, the evaluator needs fewer tests for her to reach a belief level of 99% for a truly unexploitable software, but the average number of tests for her to find a proper injection vector for a truly exploitable software changes little*. Equations (8) and (9) can be used again to explain the smaller number of tests needed to reach a certain belief level for a truly unexploitable software. When the mode of an exploitation task changes from pre-exploitation to post-exploitation, having a short memory has the following effect for any tool that is previously used for this task:

- If the observation in that test was E_0, having a short memory omits moving 1 from c_0 to c_2. This makes β/α smaller based on Eq. (8), which increases the evaluator's



(1) Left case

(2) Right case

Fig. 7. Effects of dependency. The left case shows the average number of tests needed to reach a belief level of 99% for a truly unexploitable software, and the right case the average number of tests needed to find an exploit for a truly exploitable software.

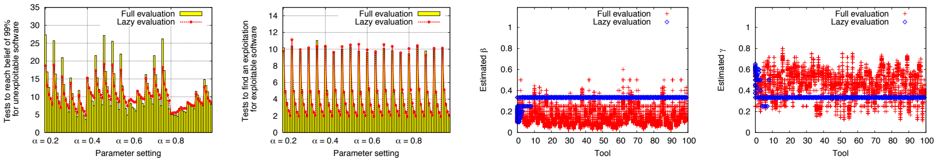
belief level with a new observation E_0 according to Eq. (1), but makes $(1-\beta-\gamma)/(1-\alpha)$ larger due to Eq. (9), which decreases the evaluator's belief level with a new observation E_2 according to Eq. (3).

- If E_2 was observed in that test, having a short memory omits moving 1 from c_1 to c_4. This makes β/α larger based on Eq. (8), which decreases the evaluator's belief level with a new observation E_0 due to Eq. (1), but makes $(1-\beta-\gamma)/(1-\alpha)$ smaller due to Eq. (9), which improves the evaluator's belief level with observation E_2 due to Eq. (3).

At first glance, having a short memory has mixed effects on a latter observation, be it E_0 or E_2. However, the key observation here is that *the impact of having a short memory on improving the evaluator's belief level is positive if the same type of observation is made later, and is negative otherwise*. Hence, if the distribution of observations is stationary over time as assumed in the experiments, the positive impact outweighs the negative one. This resembles the positive externality in economics. Therefore, having a short memory helps improve the convergence of the evaluator's belief level when the software is truly unexploitable. On the other hand, as having a short memory does not affect the estimation of parameter γ, the average number of tests to find a proper injection vector for a truly exploitable software, which is $1/\gamma$, is not affected by a short memory in parameter updating.

Effects of dependency. In another set of experiments, we evaluate effects of dependency on the evaluator's belief convergence. To model the dependency among the test results, we use the first tool to test a software independently. For any other tool, with probability p the test result is exactly the same as that done by the first one, and with probability $1-p$ the result is independent of those from the other tests. We vary dependence parameter p among 0.0, 0.2, and 0.4. Figure 7 gives how the average number of tests needed to reach a belief level of 99% for a truly unexploitable software (left) and the average number of tests needed to find an exploit for a truly exploitable software (right) change with parameter p.

Clearly, *when the test results by the tools become more similar, the evaluator needs to perform more tests to reach the same belief level for a truly unexploitable software, and also more tests to find an exploit for a truly exploitable software.* To explain this phenomenon, we examine the distribution of observations per software when $\alpha = 0.4$, $\beta = 0.2$, and $\gamma = 0.2$. As the parameter setting is the same for all the tools, we find that the total number of observations of each type (E_0, E_1, or E_2) over all software is



(1) Left Case (2) Right Case (3) β (4) γ

Fig. 8. Comparison of lazy evaluation with full evaluation (1,2) and estimation of parameters β and γ (3,4). In lazy evaluation, the evaluator stops testing a software after an exploit has been found. In contrast, full evaluation tests a software with all the tools.

similar. However, when $p = 0.4$, the distribution of these observations per software is more bursty than that when $p = 0.0$. That is to say, when $p = 0.4$, the variation of the numbers of the same type of observations is higher across different software than that when $p = 0.0$.

Different types of observations increases (or decreases) the evaluator's posterior belief to different degrees. For example, when $\beta/\alpha > (1-\beta-\gamma)/(1-\alpha)$ or equivalently, $\beta > \alpha(1-\gamma)$, the evaluator's posterior belief after seeing E_0 is lower than that after seeing E_2. As the rule of updating posterior beliefs is *nonlinear*, the average number of tests required to reach a certain belief level on a truly unexploitable software, or to find an exploit for a truly exploitable software, is not the same if we skew the distribution of different types of observations among different software even though the total numbers of observations for the same types of observations remain the same among all software.

Effects of lazy evaluation. In this set of experiments, the reachability tests are first ordered by software and then by tools. There are 100 tools and 100 software to be exploited. We model a "lazy" evaluator who, after observing the software is exploitable (i.e., seeing E_1), stops using the remaining tools to test it.

Figure 8(1,2) shows the average number of tests needed for the evaluator to reach a belief level of 99% for a truly unexploitable software and the average number of tests to find an exploit for a truly exploitable software. The parameters in the plots are ordered first by α, then β, and lastly γ. According to Figure 8(2), lazy evaluation does not affect the number of tests to find an exploitation, which is obvious as reachability tests are omitted only after the first exploit has been found for each software.

The effects of lazy evaluation on the number of tests for the evaluator to reach a belief level of 99% for a truly unexploitable software are mixed: in some cases, more tests are needed, and in others fewer are necessary. We examine the estimated values of parameters α, β and γ when their true values are 0.2, 0.1, and 0.5, respectively. Lazy evaluation does not affect much the estimation of parameter α, but it only estimates the values of parameters β and γ for a few tools, as seen in Figure 8(3,4)! That is to say, for the majority of the tools, parameters β and γ remain to be their initial values, which are 1/3 and 1/3, respectively.

The differences between lazy evaluation and full evaluation as seen in Figure 8 boil down to the differences in the estimated values of parameters β and γ. If an observation of type E_0 is seen, a larger β reduces the evaluator's posterior belief level (see Eq. (1)). On the other hand, if the new observation is of type E_2, then a larger β or γ helps improve the evaluator's posterior belief level (see Eq. (3)). With these observations, we can explain some cases where lazy evaluation requires more tests for belief convergence than full evaluation in Figure 8(1). First, when α is small, there are more observations of type E_2; as the majority of the tools in lazy evaluation have parameters β and γ set to be both 1/3, if their true values are higher than 1/3, lazy evaluation tends to underestimate their true values and thus reduces the evaluator's posterior belief level, which leads to more tests needed compared to full evaluation. The effect of parameter γ is more prominent than that of β as the latter is mixed in Eqs. (1) and (2). On the other hand, when α is large, there are more observations of type E_0. If the true value of β is smaller than 1/3, lazy evaluation always overestimates it and thus reduces the evaluator's posterior belief level according to Eq. (1), which leads to more tests needed for belief convergence than full evaluation.

8 Concluding Remarks

In this work, we propose a new cognitive framework using Bayesian reasoning as its first principle to quantify software exploitability. We rigorously analyze this framework, and also use intensive numerical simulations to study the convergence of parameter estimation and the effects of the evaluator's cognitive characteristics. In our future work, we plan to extend this work by integrating into this framework some real-world tools (e.g., software fuzzers and concolic execution tools) that can be used to exploit vulnerable software. We also plan to enrich the cognitive model used in this work.

Acknowledgment

We acknowledge the support of the Air Force Research Laboratory Visiting Faculty Research Program for this work.

References

1. Crest: Concolic test generation tool for c. In *https://jburnim.github.io/crest/*.
2. http://www.securityweek.com/shellshock-attacks-could-already-top-1-billion-report.
3. Stp constraint solver. In *http://stp.github.io/*.
4. https://nvd.nist.gov/.
5. https://www.exploit-db.com/.
6. http://www.osvdb.org/.
7. The Yices SMT Solver. In *http://yices.csl.sri.com*.
8. T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *NDSS*, volume 11, pages 59–66, 2011.
9. S. M. Bellovin. On the brittleness of software and the infeasibility of security metrics. *IEEE Security & Privacy*, (4):96, 2006.
10. D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, 2008.
11. C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
12. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

13. C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
14. CERT. Basic fuzzing framework (bff). In `https://www.cert.org/vulnerability-analysis/tools/bff.cfm?`
15. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (SP)*, pages 380–394. IEEE, 2012.
16. S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2015.
17. G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990.
18. P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.
19. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing.
20. T. L. Griffiths, C. Kemp, and J. B. Tenenbaum. Bayesian models of cognition. 2008.
21. G. Hoglund and G. McGraw. *Exploiting software: How to break code*. Addison-Wesley, 2004.
22. W. Jansen. *Directions in security metrics research*. DIANE Publishing, 2010.
23. C. Lebiere, S. Bennati, R. Thomson, P. Shakarian, and E. Nunes. Functional cognitive models of malware identification. In *Proc. of International Conference on Cognitive Modeling*, 2015.
24. P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.
25. D. McMorrow. Science of cyber-security. Technical report, JASON Program Office, 2010.
26. S. Nagaraju, C. Craioveanu, E. Florio, and M. Miller. Software vulnerability exploitation trends, 2013.
27. K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitraş. Some vulnerabilities are different than others. In *Research in Attacks, Intrusions and Defenses*, pages 426–446. Springer, 2014.
28. Forum of Incident Response and Security Teams (FIRST). Common vulnerabilities scoring system (cvss). `http://www.first.org/cvss/`.
29. A. Perfors, J. B. Tenenbaum, T. L. Griffiths, and F. Xu. A tutorial introduction to bayesian models of cognitive development. *Cognition*, 120(3):302–321, 2011.
30. A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, 2014.
31. Microsoft Research. Z3. In `https://github.com/Z3Prover/z3`.
32. S. W. Smith. Security and cognitive bias: Exploring the role of the mind. *IEEE Security & Privacy*, (5):75–78, 2012.
33. R. Telang and S. Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software Engineering*, 33(8):544–557, 2007.
34. V. Verendel. Quantified security is a weak hypothesis: a critical survey of results and assumptions. In *Proceedings of the 2009 workshop on New security paradigms workshop*. ACM, 2009.
35. A. Younis, Y. K. Malaiya, and I. Ray. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, pages 1–44.
36. C. Zhong, J. Yen, P. Liu, R. Erbacher, R. Etoty, and C. Garneau. An integrated computer-aided cognitive task analysis method for tracing cyber-attack analysis processes. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. ACM, 2015.

Appendix A: Proof of Lemma 1

We assume that before processing Q or Q', the performance counters of z_0 and z_1 are $(c_0, c_1, c_2, c_3, c_4)$ and $(c_0', c_1', c_2', c_3', c_4')$, respectively, and the priors for $\mathbb{P}\{H_0(s_0, x_0)\}$ and $\mathbb{P}\{H_0(s_1, x_1)\}$ are p_0 and p_1, respectively. We then have the following cases:

Case A: $(s_0, x_0) \neq (s_1, x_1)$ and $z_0 = z_1$. That is, the same tool is used for both exploitation tasks. In this case, the evaluator's final belief levels differ between Q and Q'. For instance, assume that $o_0 = o_1 = E_0$ and the mode is *pre-exploitation* for both tasks. For Q, we have the following:

$$\mathbb{P}\{H_0(s_0, x_0) \mid Q\} = \frac{p_0 \frac{c_0}{c_0+c_1}}{p_0 \frac{c_0}{c_0+c_1} + (1-p_0)\frac{c_2}{c_2+c_3+c_4}}$$

$$\mathbb{P}\{H_0(s_1, x_1) \mid Q\} = \frac{p_1 \frac{c_0+1}{c_0+1+c_1}}{p_1 \frac{c_0+1}{c_0+1+c_1} + (1-p_1)\frac{c_2}{c_2+c_3+c_4}}$$

By contrast, after seeing Q', we have the following:

$$\mathbb{P}\{H_0(s_1, x_1) \mid Q'\} = \frac{p_1 \frac{c_0}{c_0+c_1}}{p_1 \frac{c_0}{c_0+c_1} + (1-p_1)\frac{c_2}{c_2+c_3+c_4}}$$

$$\mathbb{P}\{H_0(s_0, x_0) \mid Q'\} = \frac{p_0 \frac{c_0+1}{c_0+1+c_1}}{p_0 \frac{c_0+1}{c_0+1+c_1} + (1-p_0)\frac{c_2}{c_2+c_3+c_4}}$$

Therefore, in this case, the cognitive model is *not* order insensitive to the tests in Q.

Case B: $(s_0, x_0) \neq (s_1, x_1)$ and $z_0 \neq z_1$. In this case, the tests are performed for different exploitation tasks using different tools. As there are no common parameters shared by the calculation of $\mathbb{P}\{H_0(s_0, x_0)\}$ and $\mathbb{P}\{H_0(s_1, x_1)\}$, the cognitive model is order insensitive in this case.

Case C: $(s_0, x_0) = (s_1, x_1)$ and $z_0 \neq z_1$. In this case, for the same exploitation task, two different tools are used. Depending on the observations o_0 and o_1, we have the following subcases.

(C.1) Either o_0 or o_1 is E_1. This means that at least one of the tools has found a successful exploit. Hence, both $\mathbb{P}\{H_0(s_0, x_0) \mid Q\}$ and $\mathbb{P}\{H_0(s_0, x_0) \mid Q'\}$ must be 0. Recall that when a successful exploitation has been found, for all the previous tests against the same exploitation task, their PCTs are revised accordingly as if the exploitation would have been found by the first tool that is ever used for the exploitation task. Hence, the final states of these tables must be the same, regardless of the order of the two tests.

(C.2) Both o_0 and o_1 are E_0. In this subcase, neither of the tools can find a proper injection vector for the exploitation task. If the mode of the task is *pre-exploitation* before the tests, we have:

$$\begin{aligned} &\mathbb{P}\{H_0(s_0, x_0) \mid Q\} = \mathbb{P}\{H_0(s_0, x_0) \mid Q'\} \\ &= \frac{p_0 \frac{c_0}{c_0+c_1} \cdot \frac{c_0'}{c_0'+c_1'}}{p_0 \frac{c_0}{c_0+c_1} \cdot \frac{c_0'}{c_0'+c_1'} + (1-p_0)\frac{c_2}{c_2+c_3+c_4} \cdot \frac{c_2'}{c_2'+c_3'+c_4'}} \end{aligned} \tag{10}$$

and after the tests, the PCTs of z_0 and z_1 are $(c_0 + 1, c_1, c_2, c_3, c_4)$ and $(c_0' + 1, c_1', c_2', c_3', c_4')$, respectively, for both Q and Q'. If the mode of the task is *post-exploitation* before the tests, the evaluator's final belief must be 0 for both Q and Q', and after the tests, the PCTs of z_0 and z_1 are $(c_0, c_1, c_2 + 1, c_3, c_4)$ and $(c_0', c_1', c_2' + 1, c_3', c_4')$, respectively, for both Q and Q'.

(C.3) Both o_0 and o_1 are 2. In this subcase, both of the tools report an exploitation with a wrong injection vector. If the mode of the task is *pre-exploitation* before the tests, we can verify that:

$$\begin{aligned}&\mathbb{P}\{H_0(s_0,x_0) \mid Q\} = \mathbb{P}\{H_0(s_0,x_0) \mid Q'\} \\ &= \frac{p_0 \frac{c_1}{c_0+c_1} \cdot \frac{c_1'}{c_0'+c_1'}}{p_0 \frac{c_1}{c_0+c_1} \cdot \frac{c_1'}{c_0'+c_1'} + (1-p_0) \frac{c_4}{c_2+c_3+c_4} \cdot \frac{c_4'}{c_2'+c_3'+c_4'}}\end{aligned} \tag{11}$$

and after the tests, the PCTs of z_0 and z_1 are $(c_0, c_1+1, c_2, c_3, c_4)$ and $(c_0', c_1'+1, c_2', c_3', c_4')$, respectively, for both Q and Q'. If the mode of the task is *post-exploitation* before the tests, the evaluator's final belief must be 0 for both Q and Q', and after the tests, the PCTs of z_0 and z_1 are $(c_0, c_1, c_2, c_3, c_4+1)$ and $(c_0', c_1', c_2', c_3', c_4'+1)$, respectively, for both Q and Q'.

(C.4) $o_0 = 0$ and $o_1 = 2$. If the mode of the task is *pre-exploitation* before the tests, we have:

$$\begin{aligned}&\mathbb{P}\{H_0(s_0,x_0) \mid Q\} = \mathbb{P}\{H_0(s_0,x_0) \mid Q'\} \\ &= \frac{p_0 \frac{c_0}{c_0+c_1} \cdot \frac{c_1'}{c_0'+c_1'}}{p_0 \frac{c_0}{c_0+c_1} \cdot \frac{c_1'}{c_0'+c_1'} + (1-p_0) \frac{c_2}{c_2+c_3+c_4} \cdot \frac{c_4'}{c_2'+c_3'+c_4'}}\end{aligned} \tag{12}$$

and after the tests, the PCTs of z_0 and z_1 are $(c_0+1, c_1, c_2, c_3, c_4)$ and $(c_0', c_1'+1, c_2', c_3', c_4')$, respectively, for both Q and Q'. If the mode of the task is *post-exploitation* before the tests, the evaluator's final belief must be 0 for both Q and Q', and after the tests, the PCTs of z_0 and z_1 are $(c_0, c_1, c_2+1, c_3, c_4)$ and $(c_0', c_1', c_2', c_3', c_4'+1)$, respectively, for both Q and Q'.

(C.5) $o_0 = 2$ and $o_1 = 0$. If the mode of the task is *pre-exploitation* before the tests, we have:

$$\begin{aligned}&\mathbb{P}\{H_0(s_0,x_0) \mid Q\} = \mathbb{P}\{H_0(s_0,x_0) \mid Q'\} \\ &= \frac{p_0 \frac{c_1}{c_0+c_1} \cdot \frac{c_0'}{c_0'+c_1'}}{p_0 \frac{c_1}{c_0+c_1} \cdot \frac{c_0'}{c_0'+c_1'} + (1-p_0) \frac{c_4}{c_2+c_3+c_4} \cdot \frac{c_2'}{c_2'+c_3'+c_4'}}\end{aligned} \tag{13}$$

and after the tests, the PCTs of z_0 and z_1 are $(c_0, c_1+1, c_2, c_3, c_4)$ and $(c_0'+1, c_1', c_2', c_3', c_4')$, respectively, for both Q and Q'. If the mode of the task is *post-exploitation* before the tests, the evaluator's final belief must be 0 for both Q and Q', and after the tests, the PCTs of z_0 and z_1 are $(c_0, c_1, c_2, c_3, c_4+1)$ and $(c_0', c_1', c_2'+1, c_3', c_4')$, respectively, for both Q and Q'.

Combining all these subcases, we can conclude that when $(s_0, x_0) = (s_1, x_1)$ and $z_0 \neq z_1$, the cognitive model is order insensitive.

Case D: $(s_0, x_0) = (s_1, x_1)$ and $z_0 = z_1$. This peculiar case deals with the situation where the same tool is used for the same exploitation task in the two tests. (1) Obviously, if $o_0 = o_1$, then the two reachability tests are indistinguishable, and the cognitive model must be order insensitive. (2) If $o_0 \neq o_1$ and one of them is 1, then for both Q and Q', the evaluator's final belief is always 0, and due to revision of PCTs after the exploitation has been found, their final states must also be the same. Hence, the cognitive model must be order insensitive. (3) If $o_0 \neq o_1$ and neither of them is 1, without loss of generality, we assume that $o_0 = 0$ and $o_1 = 2$. If the mode of the task is *pre-exploitation* before the tests, we have:

$$\begin{aligned}&\mathbb{P}\{H_0(s_0,x_0)\mid Q\}=\mathbb{P}\{H_0(s_0,x_0)\mid Q'\}\\&=\frac{p_0\frac{c_0}{c_0+c_1}\cdot\frac{c_1}{c_0+c_1+1}}{p_0\frac{c_0}{c_0+c_1}\cdot\frac{c_1}{c_0+c_1+1}+(1-p_0)\frac{c_2}{c_2+c_3+c_4}\cdot\frac{c_4}{c_2+c_3+c_4}}\end{aligned} \tag{14}$$

and the PCT is $(c_0+1, c_1+1, c_2, c_3, c_4)$ for both Q and Q'. Hence, the cognitive model is also order insensitive in this case. Otherwise, if the mode of the task is *post-exploitation* before the tests, we have $\mathbb{P}\{H_0(s_0,x_0)\mid Q\}=\mathbb{P}\{H_0(s_0,x_0)\mid Q'\}=0$, and the PCT is $(c_0, c_1, c_2+1, c_3, c_4+1)$ for both Q and Q'.

The lemma follows after combining all these four cases. □

Appendix B: Proof of Theorem 2

Proof: We need to prove that there exists a sequence of safe swappings that transforms Q to Q'. This can be done iteratively as follows. First copy Q to Q'', and during the transformation we ensure that for every reachability testing tool, the relative order of reachability tests using this tool is the same in Q' and Q''. Starting from the first element in Q', we let its index in Q'' be i_0. We can use safe swappings to move it to the head of Q''. We consider the j-th element, q'_j, in Q', which uses tool z'_j, and denote its place in Q'' as i. To use induction, we assume that the first $j-1$ elements in Q'' and Q' are the same. We can use safe swappings to move q'_j from the i-th place to the j-th place in Q'', because (1) Q'' and Q' have the same relative order of reachability tests that use the same tool, *and* (2) all the previous reachability tests that use z'_j in Q' must be within the first $j-1$ tests in Q', and thus also within the first $j-1$ tests in the current state of Q'' as the first $j-1$ tests are the same in Q' and Q''. Therefore, the first j elements of Q and Q' are now the same, and the relative order of reachability tests using any tool must still be the same in Q' and Q'' because only safe swappings are performed. By induction, after processing the $|Q'|$-th element in Q', Q'' must be the same as Q' and therefore Q can be transformed to Q' through safe swappings. □