European Journal of Operational Research 292 (2021) 596–609

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Stochastics and Statistics

To clean or not to clean: Malware removal strategies for servers under load

Sherwin Doroudi[a,*], Thanassis Avgerinos[b], Mor Harchol-Balter[c]

[a] Department of Industrial and Systems Engineering, University of Minnesota, Minneapolis, MN, USA
[b] ForAllSecure, Pittsburgh, PA, USA
[c] Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

ARTICLE INFO

Article history:
Received 28 February 2019
Accepted 27 October 2020
Available online 1 November 2020

Keywords:
Markov processes
Queueing
Computer security
Malware
Maintenance

ABSTRACT

We consider how to best schedule reparative downtime for a customer-facing online service that is vulnerable to cyber attacks such as malware infections. These infections can cause performance degradation (i.e., a slower service rate) and facilitate data theft, both of which have monetary repercussions. Infections may go undetected and can only be removed by time-consuming cleanup procedures, which require temporarily taking the service offline. From a security-oriented perspective, cleanups should be undertaken as frequently as possible. From a performance-oriented perspective, frequent cleanups are desirable because they maintain faster service, but they are simultaneously undesirable because they lead to more frequent downtimes and subsequent loss of revenue. We ask when and how often cleanups should happen.

In order to analyze various downtime scheduling policies, we combine queueing-theoretic techniques with a revenue model to capture the problem's tradeoffs. Unlike classical repair problems, this problem necessitates the analysis of a quasi-birth-death Markov chain, tracking the number of customer requests in the system and the (possibly unknown) infection state. We adapt a recent analytic technique, Clearing Analysis on Phases (CAP), to determine the exact steady-state distribution of the underlying Markov chain, which we then use to compute revenue rates and make recommendations. Prior work on downtime scheduling under cyber attacks relies on heuristic approaches, with our work being the first to address this problem analytically.

1. Introduction

Cybercrime is an increasingly costly problem for individuals, corporations, and governments alike. The total cost of cybercrime in 2018 is estimated at nearly $600 billion worldwide, amounting to about to 0.8% of the global GDP (see Center for Strategic & International Studies, 2018). In this paper, we focus in particular on cyber attacks with *persistent effects* that target an online service provider. The effects of such attacks remain until the service provider undertakes a cleanup action. These attacks often take the form of *malware* (malicious software), including viruses, code injections, Trojan horses, etc. For simplicity, throughout we will refer to these persistent attacks as malware.

Malware attacks are concerning primarily for two reasons: attacks pose a *security* threat, while also potentially compromising system *performance*; both are costly. For an online service provider, security threats include the direct theft of money as well as data breaches, which expose customers to fraud. These breaches lead to substantial legal expenses, tarnish the service provider's reputation, and where applicable, negatively affect the parent company's stock price (see Gatzlaff & McCullough, 2010). We refer to these combined monetary losses as *security losses*. Meanwhile, malware also often leads to performance degradation (see Hughes & DeLone, 2007), which effectively reduces the rate at which a service provider can serve its customers' requests (hereafter, *jobs*). Service rate reductions lead to lengthier response times, create an inferior service experience for the customers, and reduce their willingness to pay for the service. The removal of malware, although necessary, leads to additional performance costs: reboots and lengthier cleanup procedures require temporarily taking the system offline, causing discarded jobs and downtimes (see Bridwell, 2004; Diamant, Hsu, Lin, & Scoredos, 2014; Logan & Logan, 2003). Malware infections also often act as a scaffolding for even more serious malware attacks. Malware infections often develop in stages that grow worse over time, and hence security and performance costs can increase sharply if infections are left unaddressed (see Caceres, 2002;

[*] Corresponding author.
E-mail addresses: sdoroudi@umn.edu (S. Doroudi), thanassis@forallsecure.com (T. Avgerinos), harchol@cs.cmu.edu (M. Harchol-Balter).

https://doi.org/10.1016/j.ejor.2020.10.036
0377-2217/© 2020 Elsevier B.V. All rights reserved.

Huang, Arsenault, & Sood, 2006; Poolsappasit, Dewri, & Ray, 2012). More severe malware can also take longer to remove, leading to longer downtimes.

Consider a system administrator that is providing an online service. At a certain point in time, she receives an automated warning that her server is currently overloaded. This could, among other things, be a consequence of (i) a resource leak, (ii) a software bug, or (iii) a security breach that consumes system resources. The system administrator can take several actions: she could do nothing, which could have negative effects on user experience, user privacy, and ultimately revenues. Alternatively, she could reboot the system, at the cost of existing clients, with some chance of resolving the issue. Or she could take the system down, investigate what went wrong, and take time-consuming steps to maximize the likelihood that the issue is resolved. These dilemmas are faced by system administrators on a regular basis. System administrators are not the only ones that face such security-availability tradeoffs; similar dilemmas exist in every deployed system where resources are limited (see Bishop, 2012) and figuring the right balance is currently an open research question (see Darpa Cyber Grand Challenge, 2016).

In this paper we investigate how an online service provider should respond to observable changes in the system—whether *direct evidence* of an infection or performance degradation that *merely suggests the possibility* of an infection—in order to maximize revenue after accounting for security losses. In essence, we develop a mathematical model that addresses the question of “what level of threat necessitates a response?” At a first glance, it may appear that removing malware as early as possible is always revenue-optimal: early cleanups reduce average security costs, minimize performance degradation, and lead to less time-consuming cleanup times. However, frequent cleanup procedures can be detrimental, as they can increase the frequency at which jobs are discarded and lead to more downtime (even if individual downtimes are shorter).

We quantify this tradeoff by presenting a stylized but detailed Markovian stochastic model of the service provider’s operations and vulnerability to malware. Customers wait in a queue for service and pay a price that depends on the system’s historic average response time. Over time, the system becomes infected by malware in *stages*. Each successive malware state causes greater security losses, further slows down service, and takes longer to clean. Undertaking a cleanup action requires discarding all jobs currently in the system (the customers are compensated), and taking the system offline for a prolonged (random) period of time, before resuming service. Using exact stochastic analysis, we quantify the revenue rate associated with cleaning up the system at each stage and assess the performance of various cleanup policies.

In reality, malware is often difficult to detect, so we also consider the case where performance degradation can occur due to reasons other than malware, and only the service rate is visible to the service provider; the malware state is hidden. By observing the service rate, the service provider can infer probabilistic beliefs regarding malware infection and take cleanup actions based on *the duration of time spent in the current performance degradation state*.

Both the visible and hidden malware models provide an analytic challenge. In order to quantify the revenue rate under each potential policy, we must understand not only the relative proportion of time spent in each malware state, but also the mean number of jobs in the system. We determine this quantity by finding the exact limiting probability distribution of a continuous time Markov chain that simultaneously tracks the number of jobs in the system (an unbounded quantity) and the system’s current malware state. This is a *quasi-birth–death process* Markov chain with a two-dimensional state space that is infinite in one dimension. While chains of this form are notoriously difficult to analyze, we employ a novel adaptation of the Clearing Analysis on Phases (CAP) technique—developed in Doroudi, Fralix, and Harchol-Balter (2016)—to obtain the chain’s exact limiting distribution, by which we obtain exact revenues under various cleanup policies.

While problems closely related to malware cleanup have been addressed in the literature (see, e.g., Huang et al., 2006), to date work in this area has focused on heuristic solutions. We are the first to introduce and solve a mathematical model for the malware cleanup problem. Aspects of our model resemble models studied in the literature on the machine interference problem and condition-based maintenance, but with several distinguishing features (see Section 2 for details).

Our primary contributions are four-fold: (i) on the modeling front, we provide the first stochastic model for determining when an online service provider facing performance-degrading malware should perform cleanups; (ii) on the theoretical front, we derive the revenue rate for various cleanup policies in closed form—this requires solving a complex quasi-birth–death process, which requires first developing an adaptation of the CAP method; (iii) on the practical front, we provide a decision tool that allows practitioners to evaluate cleanup policies for their systems; (iv) on the case study front, we use our decision tool to highlight some interesting cases and insights by studying parameter sets provided by the security company ForAllSecure, Inc.

2. Literature review

To date, problems very closely related to the malware cleanup problem have been addressed in the computer security literature only in terms of heuristic approaches. A notable example is in the work of Huang et al. (2006), which proposes a policy where a subset of the servers being used by service provider are rotated out for cleaning, while ensuring that enough servers are running at any given time. This policy is not a result of stochastic analysis. By contrast, our model assumes a single server system, or a system where the servers work together and are prone to becoming simultaneously compromised (e.g., due to unknown exploitable bugs that are common to all servers).

The bulk of the literature on the stochastic analysis of malware and intrusions focuses on either malware propagation (see e.g., Garetto, Gong, & Towsley, 2003) or intrusion detection (see e.g., Yue & Çakanyıldırım, 2010). There is also a gradually growing body of work—including Cavusoglu, Cavusoglu, and Zhang (2008) and more recently, Bao et al. (2017)—studying security patch update management in a game-theoretic framework. These streams are not directly applicable to our problem, as our focus is on maintaining and removing potential infections from a single vulnerable system. More closely related to our work, is an emerging area of research—first explored in detail in Miehling, Rasouli, and Teneketzis (2015) and expanded upon by Backman and Ramström (2018)—employing partially observable Markov decision processes to take defense actions in the presence of potentially hidden attacks that probabilistically grow worse over time. Nonetheless our work differs markedly from this stream in that, we explicitly examine performance degradation (queueing) effects and therefore our underlying Markovian model has an infinite state space. Hence, our work necessitates different analytic machinery. Analytically, our work most closely resembles the work on machine repair (see Sections 2.1 and 2.2), along with the literature on solving quasi-birth–death process Markov chains (see Section 2.3).

2.1. Machine interference problems

The classical *machine interference* (or *repairman*) *problem*—surveyed in Haque and Armstrong (2007)—features n machines and r workers. Machines occasionally break down, and a worker can spend some time with a machine to restore it. If the number of machines that have broken down at any point exceeds the number of workers, r, then some machines will have to wait until they go into repair (i.e., the machines "interfere" with one another). When machines exhibit heterogeneity, the problem is to identify which machines should be repaired at any given time.

In contrast, our problem involves only a single server (machine), or equivalently a set of servers that work—and become compromised—together. Many successful cyber attacks involve exploiting a fundamental software-level problem, and therefore an entire system might become infected at the same time. Therefore, the malware cleanup problem asks the question of *when* to repair a system, rather than *which* subsystem to repair. Nonetheless, there is a small body of work within this stream of literature which, like our work, features potentially unbounded queues. These problems often require analyzing a quasi-birth–death process Markov chain.

Chakka and Mitrani (1994) seek to maximize the total service rate in a multi-server heterogenous queueing system. The problem's underlying Markov chain is solved *numerically* via the spectral expansion method. A similar model is considered in Wartenhorst (1995), with each server serving its own parallel queue. In Drekic and Grassmann (2002), the machines play the role of jobs in a single server queueing system, coming from two classes: low- and high-priority classes form a closed and open system, respectively. The underlying Markov chain is solved explicitly using eigenvalues.

Our Markov chain, however, has features that make it distinct from those in the works cited above. For example, our chain features infinite collections of states which transition directly to one of few states, corresponding to the fact that all customer requests (jobs) must be discarded when initiating a cleanup.

2.2. Condition-based maintenance

Condition-based maintenance problems ask *when* (i.e., *at what condition level*) failure-prone components of a system should be repaired when such components degrade in performance over time. There is a natural tradeoff between repairing components early and often (preventative maintenance) and repairing components only when necessary (corrective maintenance). The former can yield more frequent downtimes, while the latter leads to more severely degraded components and "unplanned downtime." For an overview of work in this area, see Alaswad and Xiang (2017).

The malware cleanup problem proposed in this paper can be thought of as a type of condition-based maintenance problem. The fundamental difference between our problem and the literature in this domain is that we consider a single component system, where the component serves jobs in a potentially unbounded queue. Therefore, our problem has an infinite state space, whereas the work in this area primarily focuses on finite state spaces. Maintenance actions (cleanup procedures) require the removal of all jobs in the queue, triggering an additional loss of revenue, which highlights the necessity of tracking the job count in the system. Approximations that forego tracking the job count are inadequate (see Doroudi, 2016). A maintenance model that—like our work—considers an infinite-state queueing system appears in a recent paper by Ejaz, Alvarado, Gautam, Gebraeel, and Lawley (2019); they study an M/G/1 system with multiple performance degradation states that can be remedied through corrective actions. As in our work, they seek to understand at what degradation level these corrective actions should be undertaken. However, all of their performance degradation states, except for the terminal state[1] allow the server to continue operating at full speed, and moreover, in their model corrective actions only cause a halt in service, whereas in our model cleanups also require discard existing jobs and barring further arrivals until cleanup completion. Furthermore, their paper does not consider hidden states.

Prior work has considered condition-based maintenance in the presence of hidden variables. For example, in Bunks, McCarthy, and Al-Ani (2000) decisions and inferences are made in a condition-based maintenance setting using a Hidden Markov Model (HMM), while Lin and Makis (2003) use a modified HMM with a completely observable failure state. In Makis and Jiang (2003), an optimal stopping time framework is used to approach a similar problem. Problems in this area are often modeled as partially observable Markov decision processes (POMDPs): examples include the work of Byon and Ding (2010), where the problem is solved using a backward dynamic programming method, and that of Naderkhani and Makis (2015), which addresses a problem where there is a cost to sampling the system state. While our model can be formulated as a POMDP (see our concluding remarks in Section 5), complexities that arise in our revenue rate function—due to dealing with an infinite state space problem—make the methods used in these papers unsuitable for our setting.

2.3. Methods for solving quasi-birth–death process Markov chains

Quasi-birth–death (QBD) processes are used to model a variety of phenomena. Despite the difficulties associated with finding limiting distributions for such chains, several techniques are available for analyzing specific subclasses of QBDs, the most common being matrix-geometric methods (see Neuts, 1981 and Latouche & Ramaswami, 1999). These methods are typically implemented numerically and do not generally allow for closed-form solutions.

Within the matrix-geometric literature, there are methods which can find closed-form solutions for special classes of QBDs. The method given in Van Houdt and van Leeuwaarden (2011) can find the limiting distribution of the chain describing our *visible* malware model, but is not directly applicable to our *hidden* model. The hidden model's QBD has a *directed acyclic graph* (DAG) phase transition structure, whereas the method in question is presented in the context of *tree-like* transitions. Outside the matrix-geometric literature, the Recursive Renewal Reward method (see Gandhi, Doroudi, Harchol-Balter, & Scheller-Wolf, 2013; Gandhi, Doroudi, Harchol-Balter, & Scheller-Wolf, 2014), can find closed-form solutions for chains with a DAG-like transitions in terms of means and transforms, but not complete distributions.

In this paper we employ the Clearing Analysis on Phases (CAP) method, first introduced in Doroudi et al. (2016). The CAP method yields exact closed-form solutions for the chain's entire limiting probability distribution by viewing each phase of a QBD as acting like an M/M/1 model with clearing events. This makes the CAP method a natural fit for our model, as it features clearing events (correspond to the start of a cleanup procedure, which requires discarding all jobs).

3. The case of visible malware

When a host becomes infected by malware, there are two possibilities: (i) the system administrator knows of the infection and needs to make a decision regarding whether a cleanup process should be initiated and (ii) the system administrator does not know that the host has been infected. In this section, we deal with the first case, where malware infections are always visible. While this case is certainly simpler, it is of interest for two reasons: (i) it provides a stylized simplification of the malware problem, which will lend itself to easier analysis and interpretation and (ii) it is an appropriate model for "unsophisticated malware," which serves as more of a nuisance, than a real threat.

[1] They call this state "catastrophic failure" and it is roughly analogous to our **dead** state.

3.1. Visible malware model

In the visible malware model, the system can be in one of four states (**s**) depending on the severity of malware present, if any: **normal** (uninfected), **bad**, **worse**, and **dead**. In each successive state, (i) security losses rise, (ii) performance (the service rate) drops, and (iii) the time required to remove malware increases. For tractability, all transition rates are Markovian (i.e., memoryless).

3.1.1. Queueing model

We consider a revenue-maximizing online service provider, facing a stream of incoming customers. Customer requests (hereafter, *jobs*) arrive to a first-come-first-serve queue according to a Poisson process with rate λ and are served by a single server[2] at some service rate μ, which can drop due to malware; service times are exponentially distributed. We let T be a job's response time (i.e., T is the duration from a job's arrival until it is served or discarded). We assume that the steady-state average response time $\mathbb{E}[T]$—which will be based on the service provider's decisions—is known to both the service provider and the customers (e.g., via historically available data).

3.1.2. Visible malware evolution

The server can become infected by malware in stages. Initially, the system is in the **normal** state, but will become infected by **bad** malware after an amount of time that is exponentially distributed with rate α_{bad}. Similarly, there are transitions from **bad** to **worse** (resp. **worse** to **dead**) with rate α_{worse} (resp. α_{dead}). We restrict attention to the case where malware infections occur *in sequential stages*; no stage can be skipped.[3] The model's accuracy can be increased by considering states beyond these four, but we maintain that using four states allows for an appropriate level of detail, given our aim of providing a stylized model.

Malware causes the service provider to incur security losses (e.g., monetary theft, data theft, loss of good will, etc.) and also causes the service rate to drop. The average rate of loss (in dollars per second) and service rate in state **s** are given by ℓ_s and μ_s, respectively (e.g., ℓ_{bad} and μ_{slow} in **bad**). Moreover, $\ell_{\text{dead}} \geq \ell_{\text{worse}} \geq \ell_{\text{bad}} \geq \ell_{\text{normal}} = 0$, while $\mu_{\text{fast}} \geq \mu_{\text{slow}} \geq \mu_{\text{slower}} > \mu_{\text{dead}} = 0$.[4]

The Markovian transition structures allow us to model the system as a continuous time Markov chain (CTMC) tracking the system's malware state and number of jobs in the system (*job count*) N. Fig. 1 shows our model's CTMC assuming no cleanup events. Note that without cleanups, the chain is non-ergodic, as the job count grows without bound once we reach the **dead** state. We next introduce the available cleanup policies, which will lead to modifications of this chain that will make it ergodic.

3.1.3. Visible malware cleanup

A system infected by malware can be restored to full speed by a cleanup procedure during which all existing jobs are discarded (refused service and refunded, see Section 3.1.4), and the system stops admitting jobs for a random duration of time. When the cleanup is complete, the system is restored to the **normal** state and resumes accepting jobs. The length of time devoted to the cleanup procedure depends on the current malware state, as more severe malware takes longer to remove. The cleanup procedure lasts an amount of time that is exponentially distributed with rate β_s when initiated in state **s**.[5] For simplicity, we assume that there are no security losses during cleanups.

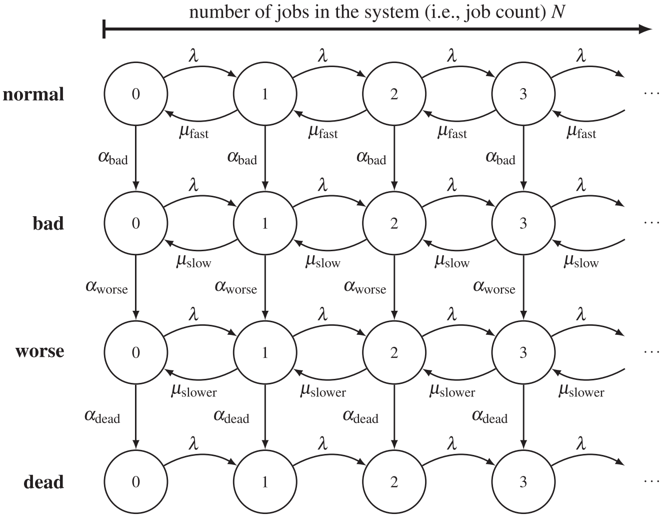

Fig. 1. The continuous time Markov chain (CTMC) tracking the malware state and the number of jobs in the system, assuming no system administration. Transitions across malware states preserve the number of jobs. The service rate drops with each successive malware state. As there are no cleanups, this chain is *not* ergodic.

In the visible malware setting, we have the following cleanup policies: **c@bad** ("clean at **bad**"), **c@worse** ("clean at **worse**"), and **c@dead** ("clean at **dead**"), which immediately clean the system when (and *only* when) the system transitions to the corresponding malware state.

3.1.4. Pricing and revenue model

We assume that customers are risk neutral and rational, opting to use the service if their service valuation equals or exceeds their total expected costs. Moreover, we assume that they are homogeneous in their service valuation q and cost of delay (in dollars per second), c. Hence, the expected cost experienced by each customer is $p + c \cdot \mathbb{E}[T]$, where p is the price (in dollars) of the service and $\mathbb{E}[T]$ is the expected response time (in seconds). Consequently, as long as the service provider charges no more than $q - c \cdot \mathbb{E}[T]$, customers will pay for the service, and since the service provider will opt to maximize its profits, they will charge precisely the maximum price that would still elicit customers to use the service, i.e., $p \equiv q - c \cdot \mathbb{E}[T]$. Should a job be discarded (either while queued or in service), the customer is refunded q (i.e., she pays nothing and keeps $q - p = c \cdot \mathbb{E}[T]$ as compensation for waiting some time before being informed that her job will not be served) and the firm incurs a revenue loss of $c \cdot \mathbb{E}[T]$ for that customer.

The service provider's objective is to implement a malware cleanup policy that maximizes the rate at which it earns revenue (less refund and security losses); we can express the service provider's revenue rate by observing that the service provider earns q for each customer that completes service, while incurring a loss of c per second for each job in the system (whether or not that customer is ultimately served; recall that customers whose jobs are not served are still compensated $c \cdot \mathbb{E}[T]$), and potentially incurring additional security losses each second. Hence, the server's revenue rate is $\mathcal{R} \equiv q\chi - c \cdot \mathbb{E}[N] - \mathcal{L}$, where q and c are as previously defined, χ is the system's throughput (i.e., the average rate

[2] The single-server assumption is for simplicity; our methodology can accommodate multiple servers.

[3] This is for expository simplicity; our methodology can accommodate skipping malware states.

[4] We assume that the system is unusable in the **dead** state, but we make no assumptions regarding the relationship between λ and the other μ_s; even if the system is "overloaded" in state **s** because $\lambda > \mu_s$, the system will still be ergodic as it will eventually be cleaned (and therefore, emptied) upon reaching the **dead** state, if not earlier. That said, in any real-world application, we would at least expect $\lambda < \mu_{\text{fast}}$.

[5] Cleanup durations can actually take on any other distribution. The analysis and results will remain unchanged, as long as the mean cleanup duration is still given by $1/\beta_s$ when initiated in state **s**.

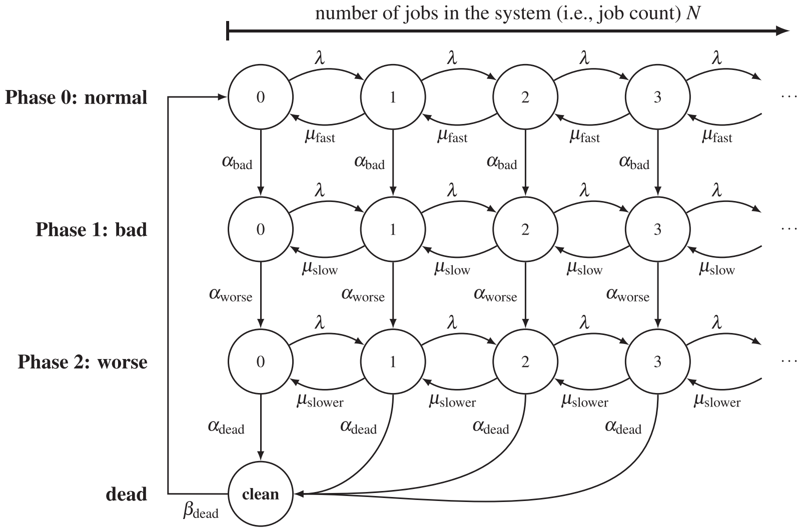

Fig. 2. The continuous time Markov chain (CTMC) tracking both the malware state and the number of jobs under the **c@dead** policy. Each *phase* (pictured as a "row" of states) is an infinite collection of states (tracking the number of jobs N) associated with one of the malware states. Transitions from one phase to another preserve the number of jobs and always move to a higher-numbered phase (pictured as "downward"). Rather than transitioning to another phase, the states in Phase 2 transition to the **clean** state, where all jobs are discarded.

at which jobs are served), $\mathbb{E}[N]$ is the steady-state time average job count, and $\mathcal{L}$ is the average rate at which security losses are incurred.

3.2. Visible malware analysis

Before analyzing $\mathcal{R}$, we define some additional notation. Let π_{normal}, π_{bad}, π_{worse}, and π_{dead} be the long-run steady-state proportion of time spent in the **normal**, **bad**, **worse**, and **dead** states, respectively, under a given policy. We can immediately observe that $\pi_{\text{bad}} = 0$ under **c@bad**, $\pi_{\text{worse}} = 0$ under both **c@bad** and **c@worse**, while $\pi_{\text{dead}} = 0$ under all policies. We can now write $\mathcal{L} = \ell_{\text{bad}}\pi_{\text{bad}} + \ell_{\text{worse}}\pi_{\text{worse}} + \ell_{\text{dead}}\pi_{\text{dead}} = \ell_{\text{bad}}\pi_{\text{bad}} + \ell_{\text{worse}}\pi_{\text{worse}}$ and consequently, $\mathcal{R} = q\chi - c \cdot \mathbb{E}[N] - \mathcal{L} = q\chi - c \cdot \mathbb{E}[N] - \ell_{\text{bad}}\pi_{\text{bad}} - \ell_{\text{worse}}\pi_{\text{worse}}$. Finding $\mathcal{R}$ under each policy requires finding χ, $\mathbb{E}[N]$, π_{bad}, and π_{worse} under that policy. While π_{bad} and π_{dead} can be found by analyzing a finite-state Markov chain tracking only the *malware state*, determining χ and $\mathbb{E}[N]$ exactly requires analyzing a two-dimensional infinite state Markov chain, which we accomplish using the Clearing Analysis on Phases (CAP) method. In the interest of brevity, we present the detailed analysis of only the most complicated policy, **c@dead**. For each policy we view our model as a CTMC capturing the malware state *and* the job count. Fig. 2 shows the **c@dead** CTMC.

Each malware state corresponds to a *phase* of the CTMC, an infinite collection of states that comprise a birth–death process, but with additional transition rates to other phases. In the **c@dead** CTMC, Phases 0, 1, and 2 correspond to the **normal**, **bad**, and **worse** malware states, respectively. We also introduce the **clean** malware state: the system is in this state when it is undergoing a cleanup procedure. The **clean** state is a single state, not a phase, as we always have $N = 0$ when cleaning.

We call attention to the fact that transitions only exist from lower-numbered phases to higher-numbered phases, which is essential for allowing the CAP method to obtain exact solutions. The CAP method treats each phase as an M/M/1 *clearing model*—an M/M/1 chain with identical rates from each state leading to a *clearing event*—where *clearing events* either lead to a higher-numbered phase or directly to the **clean** state.

In order to apply the CAP method, we introduce some notation. Let $\pi_{(m,j)}$ be the limiting probability of being in Phase m with $N = j \geq 0$ jobs in the system and let π_{clean} be the limiting probability of being in the **clean** state. Let μ_m be the Phase m service rate (i.e., $\mu_0 = \mu_{\text{fast}}$, $\mu_1 = \mu_{\text{slow}}$, $\mu_2 = \mu_{\text{slower}}$) and let α_m be the rate at which the system *leaves* Phase m (i.e., $\alpha_0 = \alpha_{\text{bad}}$, $\alpha_1 = \alpha_{\text{worse}}$, $\alpha_2 = \alpha_{\text{dead}}$).

We find limiting probabilities in the form $\pi_{(m,j)} = \sum_{k=0}^{m} a_{m,k} r_k^j$, where r_k is the *base term*[6] for Phase k,

$$r_k \equiv \frac{\lambda + \mu_k + \alpha_k - \sqrt{(\lambda + \mu_k + \alpha_k)^2 - 4\lambda\mu_k}}{2\mu_k},$$

and the coefficients $a_{m,k}$ $(0 \leq k \leq m \leq 2)$ are values associated with the relationship between Phases m and k. This form is convenient for computing $\mathbb{E}[N]$ and χ via geometric series.

Determining the limiting distribution of the **c@dead** CTMC requires only determining the $a_{m,k}$ coefficients, together with π_{clean}. These variables, together with the redundant $\pi_{(m,0)}$ variables, are the solutions to a system of linear equations, **VS**, which are a combination of balance equations and relationships which are derived via the CAP method. In Doroudi et al. (2016), the CTMCs of interest do not feature clearing events which return directly to the non-repeating portion (in the case of the **c@dead** policy, these are the transitions with rate α_{dead} from Phase 2 to the **clean** state), although it is mentioned that the CAP method can be extended to cover such transitions. We adapt the CAP method to allow it to apply to such CTMCs by modifying the balance equation associated with π_{clean} to take into account the additional transitions into π_{clean} and by considering α_{dead} as a component of α_2, the total outgoing transition rate leaving Phase 2 (in this case, we have $\alpha_2 = \alpha_{\text{dead}}$). Closed-form solutions for the limiting probability distribution of the CTMC can be obtained by symbolically solving **VS** for the $a_{m,k}$, π_{clean}, and $\pi_{(m,0)}$ variables, although the resulting expressions will not be concise. Most importantly, the solutions will be exact, rather than approximations. Applying our adaptation of the CAP method (see Appendix A in the online supplement for de-

[6] Our method requires that all r_k be distinct, which is the case for all but a zero measure set of parameter settings.

tails), the system, **VS**, is as follows:

$$\begin{cases} a_{0,0} = \pi_{(0,0)} & \pi_{(0,0)} = \frac{\beta_{\text{dead}}\pi_{\text{clean}}+\mu_0 r_0 a_{0,0}}{\lambda+\alpha_0} \\ a_{1,0} = \frac{r_0 r_1 \alpha_0 a_{0,0}}{(\lambda-\mu_1 r_0 r_1)(r_0-r_1)} & \pi_{(1,0)} = \frac{\mu_1(r_0 a_{1,0}+r_1 a_{1,1})+\alpha_0\pi_{(0,0)}}{\lambda+\alpha_1} \\ a_{1,1} = \pi_{(1,0)} - a_{1,0} & \pi_{(2,0)} = \frac{\mu_2(r_0 a_{2,0}+r_1 a_{2,1}+r_2 a_{2,2})+\alpha_1\pi_{(1,0)}}{\lambda+\alpha_2} \\ a_{2,0} = \frac{r_0 r_2 \alpha_1 a_{1,0}}{(\lambda-\mu_2 r_0 r_2)(r_0-r_2)} & \pi_{\text{clean}} = \frac{\alpha_2}{\beta_{\text{dead}}}\left(\frac{a_{2,0}}{1-r_0} + \frac{a_{2,1}}{1-r_1} + \frac{a_{2,2}}{1-r_2}\right) \\ a_{2,1} = \frac{r_1 r_2 \alpha_1 a_{1,1}}{(\lambda-\mu_2 r_1 r_2)(r_1-r_2)} & 1 = \pi_{\text{clean}} + \frac{a_{0,0}+a_{1,0}+a_{2,0}}{1-r_0} + \frac{a_{1,1}+a_{2,1}}{1-r_1} + \frac{a_{2,2}}{1-r_2} \\ a_{2,2} = \pi_{(2,0)} - a_{2,0} - a_{2,1}. & \end{cases}$$

With the limiting probabilities determined in a convenient (if not concise) form, we can compute $\mathbb{E}[N]$ and χ in terms of π_{clean} and the $a_{m,k}$ coefficients. Straightforward sum identities yield

$$\mathbb{E}[N] = \sum_{j=0}^{\infty} j \cdot \mathbb{P}(N=j) = \frac{(a_{0,0}+a_{1,0}+a_{2,0})r_0}{(1-r_0)^2} + \frac{(a_{1,1}+a_{2,1})r_1}{(1-r_1)^2} + \frac{a_{2,2}r_2}{(1-r_2)^2}.$$

Next, we compute χ, the rate at which jobs are served, *excluding* jobs that are *discarded* due to being in the system at the start of a cleanup. Since jobs only arrive when the system is in a non-**clean** state, the average arrival rate is $\lambda(1-\pi_{\text{clean}})$. Moreover, every job arriving to the system is eventually either served or discarded, so $\chi = \lambda(1-\pi_{\text{clean}}) - \eta$ where η is the discard rate. To compute η, we observe that since jobs are only discarded upon transitions to **clean**, η is the rate of such transitions **clean** (i.e., $\alpha_{\text{dead}} \cdot \pi_{\text{worse}}$) multiplied by the time average of the number of jobs at such times (i.e., $\mathbb{E}[N|\textbf{worse}]$). Hence, we have

$$\begin{aligned} \chi &= \lambda(1-\pi_{\text{clean}}) - \eta = \lambda(1-\pi_{\text{clean}}) - \alpha_{\text{dead}}\pi_{\text{worse}} \cdot \mathbb{E}[N|\textbf{worse}] \\ &= \lambda(1-\pi_{\text{clean}}) - \alpha_{\text{dead}} \sum_{j=0}^{\infty} j\pi_{(2,j)} \\ &= \lambda(1-\pi_{\text{clean}}) - \alpha_{\text{dead}}\left(\frac{a_{2,0}r_0}{(1-r_0)^2} + \frac{a_{2,1}r_1}{(1-r_1)^2} + \frac{a_{2,2}r_2}{(1-r_2)^2}\right). \end{aligned}$$

Finding $\mathcal{R}$ also requires finding the likelihood of being in each malware state: $\pi_{\text{bad}} = \sum_{j=0}^{\infty} \pi_{(1,j)} = \frac{a_{1,0}+a_{1,1}}{1-r_1}$ and $\pi_{\text{worse}} = \sum_{j=0}^{\infty} \pi_{(2,j)} = \frac{a_{2,0}+a_{2,1}+a_{2,2}}{1-r_2}$. Finally, we have

$$\begin{aligned} \mathcal{R} &= q\chi - c \cdot \mathbb{E}[N] - \ell_{\text{bad}}\pi_{\text{bad}} - \ell_{\text{worse}}\pi_{\text{worse}} \\ &= \lambda q(1-\pi_{\text{clean}}) - q\alpha_{\text{dead}}\left(\frac{a_{2,0}r_0}{(1-r_0)^2} + \frac{a_{2,1}r_1}{(1-r_1)^2} + \frac{a_{2,2}r_2}{(1-r_2)^2}\right) \\ &\quad - c\left(\frac{(a_{0,0}+a_{1,0}+a_{2,0})r_0}{(1-r_0)^2} + \frac{(a_{1,1}+a_{2,1})r_1}{(1-r_1)^2} + \frac{a_{2,2}r_2}{(1-r_2)^2}\right) \\ &\quad - \ell_{\text{bad}}\left(\frac{a_{1,0}+a_{1,1}}{1-r_1}\right) - \ell_{\text{worse}}\left(\frac{a_{2,0}+a_{2,1}+a_{2,2}}{1-r_2}\right), \end{aligned}$$

an exact determination of $\mathcal{R}$ under the **c@dead** policy, again in terms of π_{clean} and the $a_{m,k}$ coefficients.

This exact expression for the revenue rate $\mathcal{R}$ allows us to investigate the effect of various parameters on both revenue and on the optimal choice of cleanup policy. It turns out that for nearly all "realistic" parameter sets examined, **c@bad** outperforms **c@worse**, which outperforms **c@dead**. We have only found these trends to be reversed under artificial and unrealistic parameter configurations. Essentially, if you know you have been infected, you should clean the system as soon as possible. This observation validates common practices among practitioners when responding to a known infection. The best courses of action under hidden malware, however, will be much more subtle and complex.

4. The case of hidden malware

In this section we consider hidden malware, which is detectable only in the **dead** state. This case is of interest because it more accurately models the types of malware that pose serious threats to a system's security. Attackers often want their attacks to go undetected, but even stealthy malware attacks can have observable effects on the system, including *performance degradation*. While the service provider cannot know whether they have been infected by malware or not, they can still monitor their system and observe performance degradation. However, malware is not the only reason that a system may be suffering from performance degradation; the degradation may be due to memory (or other resource) leaks, locking bugs, outdated software, or other external factors. For this reason, *we model performance degradation and malware separately* (for a discussion of performance anomalies and some ways they can be mitigated, see Tan et al., 2012). At any given time, the system's *performance state* is *observable* but its *malware state* is *unobservable*.

The remainder of this section is organized as follows: we first cover the Hidden Malware Model—including some basic cleanup policies—in Section 4.1. Next, we temporarily defer the presentation of our analysis, and instead in Section 4.2, we consider a case study where we evaluate the performance of these basic policies. Guided by the insights we discover, we also propose and explore the performance of several additional policies throughout the subsection. Finally, in Section 4.3 we present the analysis of cleanup policies in the case of hidden malware that enabled our findings in our case study.

4.1. Hidden malware model

In the hidden malware model, as in the visible malware model, the system can be in one of four malware states: **normal** (uninfected), **bad**, **worse**, and **dead**, with each successive malware state leading to higher security costs and slower performance. Additionally, a non-**dead** system can be in one of three *performance* states: **fast**, **slow**, and **slower**, serving jobs at rates μ_{fast}, μ_{slow}, and μ_{slower}, respectively. While the performance state can be observed, the malware state is unobservable unless the system is **dead**. However, there is a correlation structure between the performance and malware states.[7] In particular, we assume that due to the resource-hungry nature of malware,[8] a **fast** system is necessarily **normal**, but a **slow** system can be either in the **normal** or **bad** state, and a **slower** system can be in any of the non-**dead** states.

The queueing, pricing, and revenue models are identical to those in the visible malware case; the objective remains to maximize $\mathcal{R} = q\chi - c \cdot \mathbb{E}[N] - \mathcal{L} = q\chi - c \cdot \mathbb{E}[N] - \ell_{\text{bad}}\pi_{\text{bad}} - \ell_{\text{worse}}\pi_{\text{worse}}$. While the service rate of the system depends on the system's *performance state*, only actual malware (and not performance) contributes to security loss (i.e., $\mathcal{L}$ depends directly on π_{dead} and π_{bad} rather than on π_{slow} and π_{slower}).

4.1.1. Hidden malware evolution

The server can become infected by malware and/or degrade in performance in stages, as shown in Fig. 3 (note that the chains of interest must still track the number of jobs), but only the performance state is observable. Initially, the system is **normal** and **fast**. A system in the **fast** state experiences non-malware performance degradation that causes it to become **slow** (after an exponentially distributed duration of time) with rate γ_{slow}, and a **slow** system becomes **slower** with rate γ_{slower} in the same way. A system also evolves form **normal** to **bad** to **worse** to **dead** due to malware (after an exponentially distributed duration of time) with rates α_{bad}, α_{worse}, and α_{dead}, respectively. If a **fast** system just became **bad**,

[7] By contrast, in the visible malware model a **fast**, **slow**, or **slower** system is *always* **normal**, **bad**, or **worse**, respectively.

[8] Note that our framework can also model fully stealthy malware with no performance degradation. In this paper, we specifically investigate malware that has a performance degradation effect.

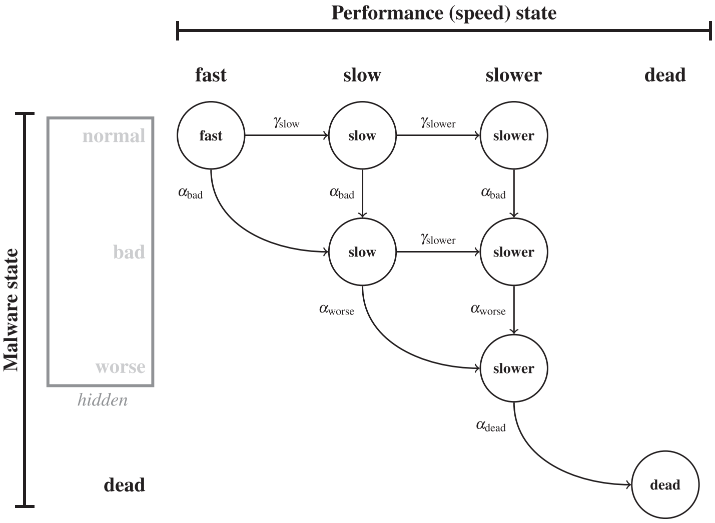

Fig. 3. The CTMC for the evolution of performance degradation and malware infection on a system under the hidden malware model without system administration. The number of jobs has been omitted, but can be viewed as being tracked by states coming "out of the page" in the third-dimension.

the resource-hungry nature of the malware will cause it to immediately become **slow** (a **slow** or **slower** system does not change speed when it becomes **bad**).[9] Similarly, if a **slow** system just became **worse**, it immediately becomes **slower**. Unlike the other malware states, the **dead** state is *not* hidden.

4.1.2. Hidden malware cleanup

A system potentially infected by hidden malware can be purged of malware (if any) and restored to full speed by a cleanup procedure. As before, such a cleanup procedure requires that all existing jobs are discarded from the system (customers are refused service and refunded q), and the system stops admitting customers for a duration of time until the system is restored to the **normal** and **fast** state. The length of time devoted to the cleanup procedure depends on the worst possible malware that *could* have infected the system—based on our model—and hence, this duration actually depends on the *performance* state. We call this the *pessimistic cleanup assumption* (e.g., a **normal slower** system takes the same amount of time to clean as a **worse slower** system). The cleanup procedure lasts an amount of time that is exponentially distributed with rate β_{bad}, β_{worse}, or β_{dead} when initiating the procedure in the **slow**, **slower**, or **dead** state, respectively;[10] that is, due to the pessimistic cleanup assumption, we clean a **slow** system as if it was **bad** and a **slower** system as if it was **worse**. We again assume that there are no security losses during cleanup.

In the setting with hidden malware, we can again consider the cleanup policies that we examined in the case of visible malware, but modifying them to respond to a change in the *performance* state, rather than the now unobservable *malware* state. This results in the **c@slow**, **c@slower**, and **c@dead** policies, which clean the system immediately upon a transitioning to the **slow**, **slower**, or **dead** state, respectively.

4.2. Hidden malware case study

In this section, we use our ability to compute exact revenue rates under the hidden malware model in order to evaluate and compare various cleanup policies. Due to our large parameter space (14 dimensions after normalizing time and money), we cannot exhaustively study the impact of all parameters on $\mathcal{R}$. While no single parameter set is representative of all systems, after consulting with the security company ForAllSecure, Inc., we chose a "default" parameter set that would be realistic in actual deployments with relatively frequent degradation and infection events. The default parameter set—denoted by **P** and presented in Table 1—allows us to keep most parameters fixed while varying one or two parameter at a time in order to observe the impact of various real-world phenomena on our system. Due to substantial security costs, **c@dead** performs poorly on all realistic cases, so for simplicity, we omit this policy from our results figures.

Our case study will focus on answering the following questions:

1. Can we gain more by improving cleanup speeds or improving intrusion detection?
2. Should we act immediately upon a performance degradation event or delay our cleanup actions?
3. What do we gain from incorporating queue length (job count) information into cleanup decisions?

In answering these questions in this case, we draw several insights regarding the malware cleanup problem.

4.2.1. Improving cleanup speeds vs. improving intrusion detection

There are several ways in which a service provider can invest resources into delivering a more robust service with the hopes of generating greater revenue. In the setting of hidden malware, two such avenues of improvement are (i) *improving cleanup speeds* (e.g., by hiring additional staff when a potential problem is identified, or by automating more steps associated with the cleanup procedure) and *improving intrusion detection* (e.g., by developing or purchasing an intrusion detection software which can reliably inform the system administrator of an attack). We explore the benefits that arise from both of these approaches.[11]

In order to quantify the benefits of improving cleanup speeds under the **P** parameter set, we leave β_{bad} fixed at 10^{-2} per second, and let $\beta_{\text{worse}} = \beta_{\text{bad}}/z$ and $\beta_{\text{dead}} = \beta_{\text{worse}}/z = \beta_{\text{bad}}/z^2$, and subsequently evaluate $\mathcal{R}$ (under both the **c@slow** and **c@slower** policies) as the free parameter z varies from 1 to 21. The lower z

[9] If a system is already **slow** when it becomes **bad**, we assume that the system is slow enough to obfuscate the impact of malware on performance, and hence does not cause a drop in performance. Alternatively, we could model this complexity with additional states if desired.

[10] As in the case of visible malware, the analysis and results remain unchanged if the cleanup durations are drawn from non-exponential distributions with the same means.

[11] For simplicity, we do not make claims about how much such improvements cost and to what extent they are feasible.

Table 1
The default parameter set **P** used throughout our case study. Recall the various families of parameters: λ (arrival rate), μ (service rates), α (malware infection rates), γ (non-malware degradation rates), β (cleanup rates), q (price of hypothetical delay-free service), c (waiting cost rate), and ℓ (security loss rates).

Default parameter set																
	1 per second												$	$ per second		
	λ	μ_{fast}	μ_{slow}	μ_{slower}	α_{bad}	α_{worse}	α_{dead}	γ_{slow}	γ_{slower}	β_{bad}	β_{worse}	β_{dead}	q	c	ℓ_{bad}	ℓ_{worse}
P	10	30	14	10.1	10^{-3}	10^{-4}	10^{-7}	10^{-2}	10^{-3}	10^{-2}	10^{-3}	10^{-4}	1	0.05	0.5	5

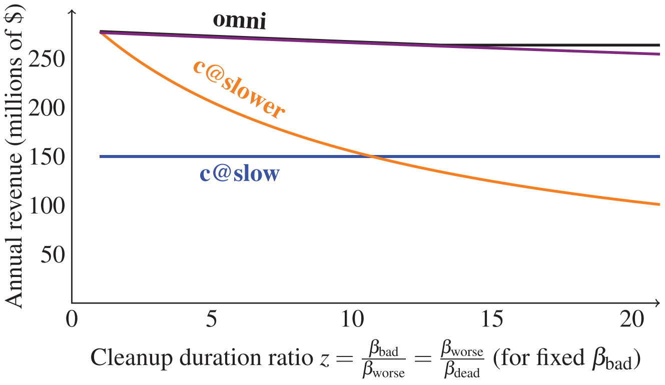

Fig. 4. Revenue rate $\mathcal{R}$ under default parameter set **P**, as a function of z, where $z = \beta_{\text{bad}}/\beta_{\text{worse}} = \beta_{\text{worse}}/\beta_{\text{dead}}$, with β_{bad} kept fixed at its default value of 10^{-2}. The **c@slow** policy exhibits constant performance (as it only depends on the fixed cleanup rate β_{bad}), while **c@slower** exhibits a convex decline, outperforming **c@slow** for low z. The hypothetical **omni** policy outperforms all other policies. The unlabeled curve is a hypothetical policy that attempts to mimic **omni** by using sequential cleanups.

is, the faster a **slower** (or **dead**) system can be cleaned. In particular, $z = 10$ corresponds to the default cleanup rates under **P** without modifications. Therefore, $z \ll 10$ corresponds to a significant investment in improving cleanup speeds. Meanwhile, we quantify the *maximum possible benefits* from improving intrusion detection by also evaluating $\mathcal{R}$ under a hypothetical policy, which we call **omni** (the "omniscient" policy) across the same range of values for z. This is a policy that can (i) actually observe the malware state of the system, (ii) ignore the pessimistic cleaning assumption, and (iii) choose an optimal subset of joint malware-performance states on which to initiate cleanups; a detailed description of this hypothetical policy is provided in Appendix B in the online supplement. The revenue rates are plotted in Fig. 4.

We first observe that under **c@slow**, $\mathcal{R}$ is constant in z. This is because under **c@slow**, one never reaches (let alone initiates a cleanup in) the **slower** or **dead** state; hence, $\mathcal{R}$ depends only on β_{bad} and not on β_{worse} or β_{dead}. Meanwhile, **c@slower** outperforms **c@slow** for low values of z, exhibiting a convex decline. This shape is apparent across a wide range of realistic system parameters (and it can be proven to be a hyperbola). We also observe that at $z = 1$, $\mathcal{R}$ under **c@slower** matches that under **omni**. To understand why, we note that **omni** behaves like **c@slower** for low values of z, except that **omni** circumvents the pessimistic cleaning assumption (it cleans based on the actual *malware* state rather than on *performance* state). At $z = 1$, however, **omni** and **c@slower** achieve the same revenue rate, because cleaning a **slower** system does not take less time if one is aware of the kind of malware present (if any) when $\beta_{\text{bad}} = \beta_{\text{worse}}$ (i.e., when $z = 1$). Eventually (visible in the figure at $z \approx 13.2$), the **omni** policy will clean a system as soon as it is **slower** or **bad**. We can conclude that intrusion detection can significantly improve the profitability of a system, but so can reducing the time required to cleanup more serious problems on a system. In this case, the benefits from *perfect* intrusion detection outweigh those from all but the most extreme improvements in cleanup speeds, suggesting that improving intrusion detection should be a higher priority.

In fact, it turns out that much of the benefit in the hypothetical **omni** policy is due to the fact that it is not bound by the pessimistic cleaning assumption (i.e., it does not need to implement a lengthy cleanup procedure if a **slower** system is not in the **worse** malware state). It can be tempting to mimic this advantage even when malware is not observable by cleaning a **slower** system by using a shorter cleanup (i.e., with cleaning rate β_{bad}). Then, if at the conclusion of that cleanup the system is still sluggish (observed to be in the **slower** state due to lingering malware that was not removed by the quick cleanup), one can implement a lengthier cleanup (i.e., with cleaning rate β_{worse}) that is guaranteed to remove the malware. The impressive performance of this "sequential" cleanup policy is shown by the unlabeled curve in Fig. 4. Unfortunately, such a policy may not always be implementable in practice, as it might be a poor security practice to perform an insufficient cleanup procedure when there is reason to suspect a serious infection. However, whenever such a policy is considered "safe enough," it is a strong alternative to improving intrusion detection. For example, before formatting a system that appears to be potentially infected, it may pay off to perform a quick reboot to see if the problem persists. In practitioners' terms, this is "trying the easy solution first."

We conclude that for our case study, intrusion detection is preferable to improving cleanup speeds (if one must choose only one and the two improvement costs are comparable) whenever near-perfect intrusion is possible, unless it is safe to use a sequence of progressively lengthier cleanups when dealing with a system in the **slower** state. However, we must acknowledge that the **omni** policy is a *hypothetical* policy, and that perfect intrusion detection cannot exist in practice. Nevertheless, while we cannot actually detect malware perfectly, the fact that—in many typical parameter sets such as **P**—hidden malware infection occurs at a considerably lower frequency than natural performance degradation should not be overlooked. We proceed to develop a family of policies that can leverage this fact.

4.2.2. Delaying cleanups

It is natural to ask whether a system should be cleaned as soon as one reaches a "target" performance degradation state, or if the cleanup procedure should be delayed once such a "target" state has been reached. The rationale for such a delay is that the service provider may be content with slower performance, but *not* with malware infections. In particular, since $\alpha_{\text{bad}} = 10\gamma_{\text{slow}}$ under **P**, a transition to the **slow** state is unlikely to suggest that the system has been infected by **bad** malware, and one can persist in a **slow** system for a considerable period of time with little risk of being unknowingly infected. But how much of an improvement in $\mathcal{R}$ can we expect if we introduce such delays? In order to explore such delays, we formally introduce the following two parameterized families of policies:

- **dc@slow**(ξ) ("delayed clean at **slow**" with rate ξ): After the system transitions to the **slow** state, wait an amount of time that is exponentially distributed with rate ξ, then clean; clean immediately upon transitioning to **slower**.

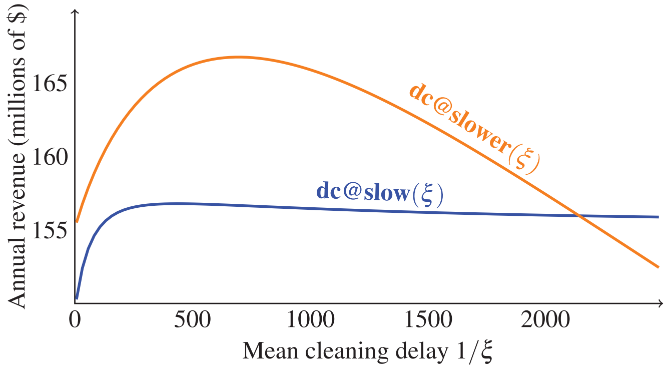

Fig. 5. Revenue rate $\mathcal{R}$ under the default parameter set **P**, as a function of the mean delay before initiating cleanup actions $1/\xi$. The revenue rate $\mathcal{R}$ appears to be concave in $1/\xi$, with both policy families benefiting from the introduction of a modest delay. However, under both policy families, $\mathcal{R}$ eventually attains a local maximum in $1/\xi$, and there is a subsequent decrease in profitability as $1/\xi$ increases further. At their respective optimal ξ values, **dc@slower**(ξ) outperforms **dc@slow**(ξ).

- **dc@slower**(ξ) ("delayed clean at **slower**" with rate ξ): After the system transitions to the **slower** state, wait an amount of time that is exponentially distributed with rate ξ, then clean; clean immediately upon transitioning to **dead**.

If the delays were *deterministic* rather than random, one could potentially obtain even greater revenue rates: a random delay is a lottery over a continuum of deterministic delays (with a potentially new outcome after every cleanup event), and so the best policy among the support of this lottery will do no worse than the lottery itself. We restrict attention to exponentially distributed delays for the purpose of tractability, in order to maintain a Markovian structure.[12]

We proceed to explore the benefits of implementing such delays by evaluating $\mathcal{R}$ under the **dc@slow**(ξ) and **dc@slower**(ξ) policies for the parameter set **P**. Fig. 5 shows a comparison of the revenue rates under these cleanup policies as a function of the mean cleanup delay $1/\xi$. We observe that for this parameter set, there is a significant benefit to implementing a delay under both policies. In fact, introducing such delays is beneficial in nearly all systems, except those where performance degradation and/or malware is so costly that these costs dwarf the detrimental impact of frequent cleanups. Here, the best performing policy is **dc@slower**$(1/696)$, attaining a revenue rate of $\mathcal{R} \approx 166.74$ (measured in millions of dollars per year). The sensitivity of these results is explored in Appendix C in the online supplement. While we anticipate robustness to small measurement or estimation errors in a single parameter (except possibly λ and μ_{slower} as they are very close to one another in the default parameter set **P**), modest to large errors in measuring γ_{slower}, β_{bad}, β_{worse}, q, or c can be costly in the sense that they can lead to the implementation of a considerably suboptimal cleanup policy.

The revenue obtained by the **dc@slower**$(1/696)$ policy can be improved further by using a delayed cleanup policy that allows cleaning in both the **slow** and **slower** states, implementing a separate delay rate at each. In more precise terms, we consider the following family of policies:

- **hdc**(ξ_1, ξ_2) ("hybrid delayed clean" with rates ξ_1 and ξ_2): After the system transitions to the **slow** state, wait an amount of time that is exponentially distributed with rate ξ_1, then clean, unless a transition to **slower** occurs, in which case, disregard the wait so far and instead wait an amount of time that is independently exponentially distributed with rate ξ_2, *then* clean the system; clean immediately upon transitioning to **dead**.

Exploring many ξ_1 and ξ_2 values, the best performance we observe under this hybrid-policy, is at $\xi_1^* = 1/680$ and $\xi_2^* = 1/749$ (we will refer to these "optimal delay rates" repeatedly in what follows). The **hdc**$(1/680, 1/749)$ policy yields $\mathcal{R} \approx 167.96$. By using delays in both the **slow** and **slower** states, we have a modest improvement of less than 1% (as compared to the simpler **dc@slower**$(1/696)$ policy), although such improvements can be more pronounced across a variety of parameter settings.

Naturally, one benefit of delaying cleanups is decreasing the frequency of cleanup actions, while adding little additional risk of residing in or entering a malware state. For example, if one has already transitioned to the **slower** state, one has already effectively "paid the sunk cost" of a lengthier cleanup duration (which is unlikely to grow any longer if one imposes a reasonable delay, as transition rates to the **dead** state are typically very low). In this case, one may as well decrease the frequency of cleanup procedures by spending additional time in the **slower** state. However, this is not the only benefit to implementing cleanup delays. When a change in *performance* level occurs, waiting times gradually increase over time, rather than increasing immediately. Therefore, if an average response time of less than t^* is "acceptable," (i.e., it is profitable to operate under such average response times, in the interest of engaging in less frequent cleanups) and one is transitioning from a performance state with a steady-state response time of $t_1 \ll t^*$ to one with $t_2 \gg t^*$, one can delay a cleanup event and still enjoy "acceptable" response times for some additional time while "spacing out" cleanups. Hence, delaying cleanups can even be beneficial in the case of visible malware. The takeaway is that we should consider *not acting immediately* upon a performance degradation event. Waiting for an appropriate amount of time can lead to significant gains in revenue.

4.2.3. Cleaning up based on queue length

In the preceding discussion, the gradual transition from one steady-state response time to a higher steady-state response time is due to the gradual buildup of the queue. Hence, motivated by the benefits that come with delaying cleanups, we consider *dynamic* cleanup policies that can base cleanup decisions on the current job count. But how much can the service provider benefit from taking queue lengths into account?

We shed light on the potential benefits of dynamic policies by evaluating $\mathcal{R}$ for the special cases of dynamic policies that we call *threshold policies*. Like the policies we have examined thus far, threshold policies also incorporate delays, but they initiate cleanups in the **slower** state only if the queue length satisfies a given condition based on a threshold, Θ. More precisely, we consider these two policy families:

- **hdc**$_{\geq\Theta}(\xi_1, \xi_2)$: After the system transitions to the **slow** state, wait an amount of time that is exponentially distributed with rate ξ_1, then clean, unless a transition to **slower** occurs. Once the total amount of time in the **slower** state with $N \geq \Theta$ jobs exceeds an (independent) exponentially distributed random variable with rate ξ_2, clean the system; if a transition to **dead** occurs, clean immediately.
- **hdc**$_{\leq\Theta}(\xi_1, \xi_2)$: This is the same as the preceding policy, except when in the **slower** state, we clean based on having spent sufficient time with $N \leq \Theta$ jobs (rather than $N \geq \Theta$ jobs).

We note that the **hdc**$_{\geq\Theta}(\xi_1, \xi_2)$ (resp. **hdc**$_{\leq\Theta}(\xi_1, \xi_2)$) policy is like the **hdc**(ξ_1, ξ_2) policy, except that the transition from the **slower** state to the corresponding cleanup state occurs with rate

[12] One could more closely approximate deterministic delays by implementing them as multi-phase Erlang distributions, which approach deterministic distributions as the number of (identical) phases tends to infinity, while keeping the mean fixed.

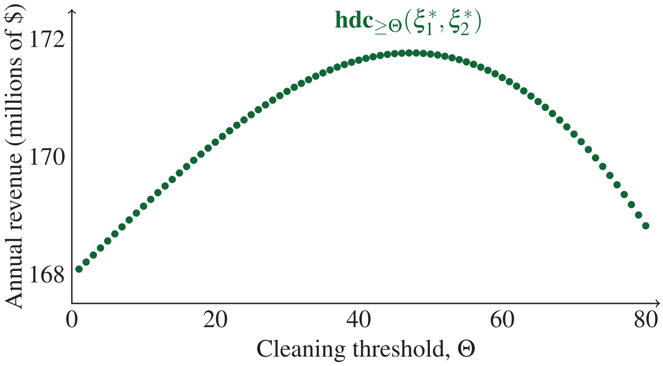

Fig. 6. Revenue rate $\mathcal{R}$ under the default parameter set **P**, as a function of the threshold Θ, for the $\mathbf{hdc}_{\geq\Theta}(\xi_1^*, \xi_2^*)$ family of policies, where cleanups in the **slower** state are only permitted if the number of jobs N exceeds the threshold Θ. $\mathcal{R}$ is highest at $\Theta = 47$.

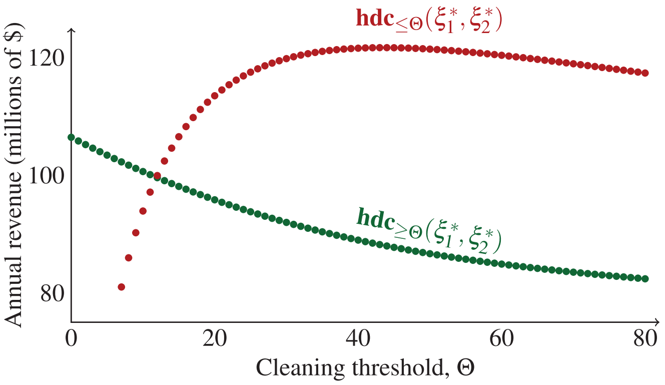

Fig. 7. Revenue rate $\mathcal{R}$ under the default parameter set **P** with a job discarding penalty of $y = \$100$, as a function of the the threshold Θ, for the $\mathbf{hdc}_{\geq\Theta}(\xi_1^*, \xi_2^*)$ and $\mathbf{hdc}_{\leq\Theta}(\xi_1^*, \xi_2^*)$ families of policies. $\mathcal{R}$ is highest at $\Theta \in \{43, 44\}$ under the latter family.

ξ_2 only when the number of jobs in the system is *at least* (resp. *at most*) Θ.

We begin by studying the $\mathbf{hdc}_{\geq\Theta}(\xi_1^*, \xi_2^*)$ family of policies across a range of Θ values, where we fix $\xi_1^* = 1/680$ and $\xi_2^* = 1/749$, which were the best rates (with respect to $\mathcal{R}$) for the non-threshold $\mathbf{hdc}(\xi_1, \xi_2)$ family of policies that we studied in the previous subsection. We note that we can view $\mathbf{hdc}_{\geq 0}(\xi_1^*, \xi_2^*)$ as a benchmark that is identical to $\mathbf{hdc}(\xi_1^*, \xi_2^*)$, the best performing policy from the previous subsection. Fig. 6 depicts the revenue rates under the $\mathbf{hdc}_{\geq\Theta}(\xi_1^*, \xi_2^*)$ policies. We observe that among the plotted policies, an optimal[13] revenue rate of $\mathcal{R} \approx 171.75$ is achieved at the threshold $\Theta = 47$, representing an improvement of more than 2% over the $\Theta = 0$ benchmark. This modest yet non-negligible improvement highlights the power of dynamic policies and suggests that cleanup delays alone are insufficient in capturing nearly all available revenue. This realization also underscores the advantage of tracking the number of jobs in the system as opposed to treating malware cleanup as a standard condition-based maintenance problem.

Let us now turn our attention to the other family of threshold policies, $\mathbf{hdc}_{\leq\Theta}(\xi_1, \xi_2)$; the policies in this family initiate cleanup procedures in the **slower** state only when the number of jobs N falls at or *below* the threshold Θ. We would expect such policies (for small values of Θ) to perform poorly for the same reasons the preceding policies performed well.[14] These alternative threshold policies allow the system to persist in the **slower** state for far too long, as the system is rarely occupied by only a few jobs in the **slower** state.

Naturally, we ask if there exist scenarios where the $\mathbf{hdc}_{\leq\Theta}(\xi_1, \xi_2)$ policies outperform their $\mathbf{hdc}_{\geq\Theta}(\xi_1, \xi_2)$ counterparts. As one can imagine, the $\mathbf{hdc}_{\leq\Theta}(\xi_1, \xi_2)$ policies excel at minimizing the number of discarded jobs when a cleaning procedure is initiated. In fact, they limit this number to Θ per cleanup event triggered in the **slower** state. It turns out that the cost associated with discarding a job under **P** is often relatively insignificant. Even if hundreds of jobs are discarded at once, the number of jobs served between cleaning procedures may be orders of magnitude higher than this figure. However, one can imagine a modified setting where there is a much stronger desire to minimize discarded jobs.

Consider a variation of the model explored in this paper where each discarded job represents more than just a missed opportunity for the service provider to collect q for serving an additional customer request. For each discarded job, the service provider incurs a goodwill penalty y (measured in dollars). That is, we could take $q\chi - c \cdot \mathbb{E}[N] - \ell_{\text{bad}}\pi_{\text{bad}} - \ell_{\text{worse}}\pi_{\text{worse}} - y\eta$, as our revenue rate (recall that η is discard rate).

Now let us consider the default parameter setting **P**, except that we will let $y = \$100$, rather than $y = \$0$ (which is the case everywhere else in this paper). Note that this is an extreme value chosen for illustrative purposes. Evaluating revenues for both the $\mathbf{hdc}_{\geq\Theta}(\xi_1^*, \xi_2^*)$ and $\mathbf{hdc}_{\leq\Theta}(\xi_1^*, \xi_2^*)$ policy families, we observe in Fig. 7 that $\mathcal{R}$ is decreasing in Θ for $\mathbf{hdc}_{\geq\Theta}(\xi_1^*, \xi_2^*)$, rendering the $\Theta = 0$ benchmark optimal among the policies in this family; this result is driven by the extreme value of y. Meanwhile, $\mathcal{R}$ is initially increasing (and subsequently decreasing) in Θ for $\mathbf{hdc}_{\leq\Theta}(\xi_1^*, \xi_2^*)$. The strongest among these policies overall (among those being considered in this setting) is either $\mathbf{hdc}_{\leq 43}(\xi_1^*, \xi_2^*)$ or $\mathbf{hdc}_{\leq 44}(\xi_1^*, \xi_2^*)$, which allows for cleanups in the **slower** state only when there are at most 43 or 44 jobs present in the system. These two policies exhibit virtually indistinguishable performance: based on our numerical calculations, they obtain $\mathcal{R}$ values that are within 0.00002% of one another. Meanwhile, both policies outperform the best performing policy from the other family, $\mathbf{hdc}_{\geq 0}(\xi_1^*, \xi_2^*)$, (i.e., the $\Theta = 0$ benchmark) by over 14%. This significant improvement suggests that in this particular setting, cleanup procedures should only be undertaken when the queue length is relatively short. In this setting, it turns out that the common intuition that “the more highly utilized a system is, the more costly it is to take it offline” is justified.

At this point, however, we should be wary of tunnel vision: with a sizable discarding penalty of $y = 100$, $1/\xi_1^*$ and $1/\xi_2^*$ may no longer be ideal mean delay candidates to build our threshold policies upon. In fact, dispensing with cleanup delays altogether we can improve upon $\mathbf{hdc}_{\leq 44}(\xi_1^*, \xi_2^*)$ substantially in this setting: employing a policy that cleans immediately whenever the job count is 0 or 1 in either the **slow** or **slower** states yields an improvement of over 23%. An even larger improvement (over 27%) is possible if we only clean a **slow** system when it is empty, but still clean a **slower** system at either job count 0 or 1.

The observation that both families of threshold policies can outperform the other depending on the setting suggests that even further gains are possible by considering more sophisticated dynamic policies. However, as previously stated, determining the optimal

[13] Naturally, we can expect to do better if we jointly optimize the delays ξ_1 and ξ_2 together with the threshold Θ, or consider dynamic policies beyond simple threshold policies. We note that allowing for two separate thresholds (below which we cannot clean), one for the system in the **slow** state and the other for the **slower** state did not appear to yield any benefits in this case.

[14] We have verified that this is the case under **P** for the $\mathbf{hdc}_{\leq\Theta}(\xi_1^*, \xi_2^*)$ family of policies.

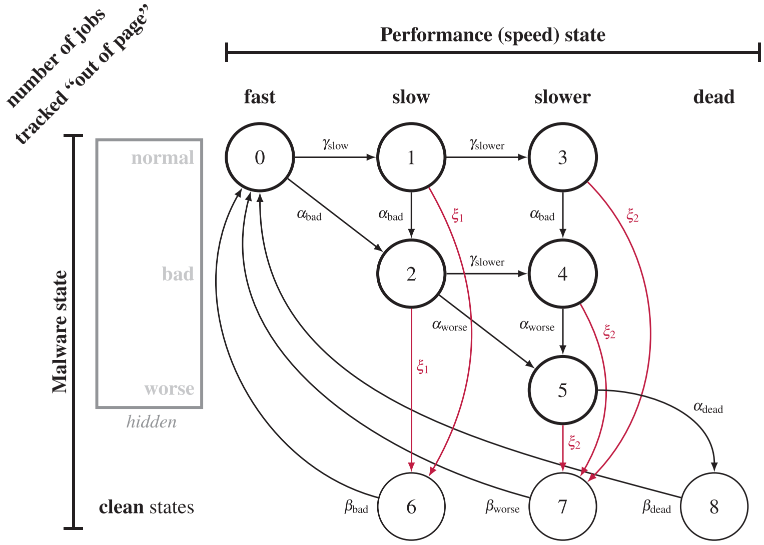

Fig. 8. The CTMC governing the system under the $\mathbf{hdc}(\xi_1, \xi_2)$ cleanup policy. States and phases are labeled by the numbers 0–8, for notational convenience, with *Phases* 0–5 denoting the six joint malware-performance states, and *states* 6–8 denoting the three possible **clean** states (**clean-short**, **clean-med**, and **clean-long**). The transitions due to intentional delays imposed by the cleanup policy are shown in color for clarity. Note that each *phase* (denoted by a thicker border) is actually an infinite collection of states that evolves according to a birth–death process.

dynamic policy requires solving an intractable dynamic program. We are able to obtain results for these threshold policies by extending the non-repeating portion of the Markov chain when applying our adaptation of the CAP method.

4.3. Hidden malware analysis

We now turn our attention to deriving $\mathcal{R}$ in the case of hidden malware under all of the policies presented throughout this section. As it turns out, we can actually restrict attention to the analysis of $\mathcal{R}$ under the $\mathbf{hdc}(\xi_1, \xi_2)$ class of policies. With the exception of the threshold policies, revenues under the other policies can be computed by taking the appropriate limits (and in the case of **omni**, by choosing the best among several candidate policies).[15] As for the threshold policies, $\mathbf{hdc}_{\geq\Theta}(\xi_1, \xi_2)$ and $\mathbf{hdc}_{\leq\Theta}(\xi_1, \xi_2)$, a modification of the procedure described in this section allows us to determine $\mathcal{R}$ under these policies in essentially the same way. This modification involves expanding the non-repeating portion of the Markov chain of interest to encompass all states where the number of jobs in the system is less than Θ.

We again need to calculate $\mathcal{R} = q\chi - c \cdot \mathbb{E}[N] - \ell_{\rm bad}\pi_{\rm bad} - \ell_{\rm worse}\pi_{\rm worse}$, χ, and $\mathbb{E}[N]$, which require the analysis of a two-dimensional infinite state Markov chain, this time with six, rather than three, phases.

The first step in our approximation relies on determining the limiting probability associated with the six joint malware-performance states (see Fig. 3) and each of three separate **clean** states: **clean-short**, **clean-med**, and **clean-long**, which return to the **normal fast** state with rates $\beta_{\rm bad}$, $\beta_{\rm worse}$, and $\beta_{\rm dead}$, respectively. For notational convenience we label the phases and states of interest with the numbers 0–8, as follows: Phase 0 corresponds to the **normal** state of a **fast** system, Phases 1 and 2 correspond to the **normal** and **bad** states of a **slow** system, respectively, while Phases 3, 4, and 5 corresponds to the **normal**, **bad**, and **worse** states of a **slower** system, respectively. Moreover, *states* 6, 7, and 8 correspond to **clean-short**, **clean-med**, and **clean-long** states, respectively.

With this notation, we proceed to apply our adaptation of the Clearing Analysis on Phases (CAP) method to the hidden malware model under the $\mathbf{hdc}(\xi_1, \xi_2)$ policy. The CTMC we study looks like a more complicated version of the chain depicted in Fig. 2 (from the analysis of the **c@dead** policy for the visible malware model), except that it consist of six *phases*, and three *cleanup states*, with some transitions across phases "skipping over" intermediate phases, and multiple phases transitioning directly to cleanup states. Our CTMC consists of an infinite *repeating portion* and a finite *non-repeating portion*.

The repeating portion of our CTMC is made up of the six phases, one corresponding to each of the joint malware-performance states, 0–5. Each phase is an infinite collection of states making up a birth–death process tracking the number of jobs in the system. Each birth–death process has an arrival rate of λ, and departure rate of $\mu_{\rm fast}$ (in Phase 0), $\mu_{\rm slow}$ (in Phases 1 and 2), or $\mu_{\rm slower}$ (in Phases 3–5). Each state in the repeating portion of the Markov chain is denoted by (m, j), where m is the phase and $N = j$ denotes the job count. We observe that the transitions across these phases are *unidirectional* in nature (i.e., when moving from one phase in $\{0, \ldots, 5\}$ to another, the phase number always increases), which makes the CTMC amenable to the CAP method. Moreover, transitions across these phases are independent of the job count, with associated transition rates depicted in Fig. 8 (e.g., a transition from Phase 2 to Phase 5 occurs with rate $\alpha_{\rm worse}$, regardless of the job count). When any such transition occurs the job count remains unchanged.

The non-repeating portion of our CTMC is made up of the three cleanup states: **clean-short** (6), **clean-med** (7), and **clean-long** (8). Unlike phases, these are *single states*. As these states represent the system undergoing a cleanup, they always transition directly to state (0,0) (i.e., Phase 0 with an empty queue). Transitions *to* one of cleanup states from one of the phases are independent of the job count, and occur with the rates given in Fig. 8. Such a transition resets the job count to zero (as the jobs are discarded).

We introduce some notation. Let $\pi_{(m,j)}$ be the limiting probability of being in Phase $m \in \{0, \ldots, 5\}$ with $j \geq 0$ jobs. Let π_6, π_7, π_8 be the limiting probabilities of being in cleanup *states* 6, 7, and 8, respectively. Let μ_m be the service rate in Phase m

[15] There can be computational advantages to determining $\mathcal{R}$ under the other policies directly (rather than taking limits of $\mathcal{R}$ under the $\mathbf{hdc}(\xi_1, \xi_2)$ policies). Such direct calculations follow from simple modifications to the analyses presented in this section.

($\mu_0 = \mu_{\text{fast}},\ \mu_1 = \mu_2 = \mu_{\text{slow}},\ \mu_3 = \mu_4 = \mu_5 = \mu_{\text{slower}}$) and let α_m be the rate at which the system *leaves* Phase m.[16]

We again find limiting probabilities in the form $\pi_{(m,j)} = \sum_{k=0}^{m} a_{m,k} r_k^j$, with base terms (which we again assume are all distinct) given by

$$r_k = \lambda + \mu_k + \alpha_k - \frac{\sqrt{(\lambda + \mu_k + \alpha_k)^2 - 4\lambda\mu_k}}{2\mu_k}.$$

We must determine the $a_{m,k}$ coefficients (for $0 \le k \le m \le 5$), together with π_6, π_7, π_8. These variables, together with the redundant $\pi_{(m,0)}$ variables, are the solutions to a system of linear equations, **HS**, which are a combination of balance equations, the normalization equation, and relationships derived via our adaptation of the CAP method (see Appendix A in the online supplement for details). The system **HS** is as follows:

$$\begin{cases}
a_{0,0} = \pi_{(0,0)} \\
a_{1,0} = \dfrac{r_0 r_1 \gamma_{\text{slow}} a_{0,0}}{(\lambda - \mu_1 r_0 r_1)(r_0 - r_1)} \\
a_{2,0} = \dfrac{r_0 r_2 \alpha_{\text{bad}} (a_{0,0} + a_{1,0})}{(\lambda - \mu_2 r_0 r_2)(r_0 - r_2)} \\
a_{2,1} = \dfrac{r_1 r_2 \alpha_{\text{bad}} a_{1,1}}{(\lambda - \mu_2 r_1 r_2)(r_1 - r_2)} \\
a_{3,k} = \dfrac{r_k r_3 \gamma_{\text{slower}} a_{1,k}}{(\lambda - \mu_3 r_k r_3)(r_k - r_3)} & (0 \le k \le 1) \\
a_{3,2} = 0 \\
a_{4,k} = \dfrac{r_k r_4 (\gamma_{\text{slower}} a_{2,k} + \alpha_{\text{bad}} a_{3,k})}{(\lambda - \mu_4 r_k r_4)(r_k - r_4)} & (0 \le k \le 2) \\
a_{4,3} = \dfrac{r_3 r_4 \alpha_{\text{bad}} a_{3,3}}{(\lambda - \mu_4 r_3 r_4)(r_3 - r_4)} \\
a_{5,k} = \dfrac{r_k r_5 \alpha_{\text{worse}} (a_{2,k} + a_{4,k})}{(\lambda - \mu_5 r_k r_5)(r_k - r_5)} & (0 \le k \le 2) \\
a_{5,3} = \dfrac{r_3 r_5 \alpha_{\text{worse}} a_{4,3}}{(\lambda - \mu_5 r_3 r_5)(r_3 - r_5)} \\
a_{5,4} = \dfrac{r_4 r_5 \alpha_{\text{worse}} a_{4,4}}{(\lambda - \mu_5 r_4 r_5)(r_4 - r_5)}
\end{cases}$$

$$\begin{aligned}
a_{m,m} &= \pi_{(m,0)} - \textstyle\sum_{k=0}^{m-1} a_{m,k} \qquad (1 \le m \le 5) \\
\pi_{(0,0)} &= \frac{1}{\lambda + \alpha_0} (\beta_{\text{bad}} \pi_6 + \beta_{\text{worse}} \pi_7 + \beta_{\text{dead}} \pi_8 + \mu_0 r_0 a_{0,0}) \\
\pi_{(1,0)} &= \frac{1}{\lambda + \alpha_1} \left(\mu_1 \textstyle\sum_{k=0}^{1} (r_k a_{1,k}) + \gamma_{\text{slow}} \pi_{(0,0)}\right) \\
\pi_{(2,0)} &= \frac{1}{\lambda + \alpha_2} \left(\mu_2 \textstyle\sum_{k=0}^{2} (r_k a_{2,k}) + \alpha_{\text{bad}} (\pi_{(0,0)} + \pi_{(1,0)})\right) \\
\pi_{(3,0)} &= \frac{1}{\lambda + \alpha_3} \left(\mu_3 \textstyle\sum_{k=0}^{3} (r_k a_{3,k}) + \gamma_{\text{slower}} \pi_{(1,0)}\right) \\
\pi_{(4,0)} &= \frac{1}{\lambda + \alpha_4} \left(\mu_4 \textstyle\sum_{k=0}^{4} (r_k a_{4,k}) + \gamma_{\text{slower}} \pi_{(2,0)} + \alpha_{\text{bad}} \pi_{(3,0)}\right) \\
\pi_{(5,0)} &= \frac{1}{\lambda + \alpha_5} \left(\mu_5 \textstyle\sum_{k=0}^{5} (r_k a_{5,k}) + \alpha_{\text{worse}} (\pi_{(2,0)} + \pi_{(4,0)})\right) \\
\pi_6 &= \frac{\xi_1}{\beta_{\text{bad}}} \textstyle\sum_{m=1}^{2} \sum_{k=0}^{m} \dfrac{a_{m,k}}{1 - r_k} \\
\pi_7 &= \frac{\xi_2}{\beta_{\text{worse}}} \textstyle\sum_{m=3}^{5} \sum_{k=0}^{m} \dfrac{a_{m,k}}{1 - r_k} \\
\pi_8 &= \frac{\alpha_{\text{dead}}}{\beta_{\text{dead}}} \textstyle\sum_{k=0}^{5} \dfrac{a_{5,k}}{1 - r_k} \\
1 &= \left(\textstyle\sum_{m=0}^{5} \sum_{k=0}^{m} \dfrac{a_{m,k}}{1 - r_k}\right) + \pi_6 + \pi_7 + \pi_8
\end{aligned}$$

This system is more complicated than the corresponding system for the case of visible malware (**VS**), as in the present model one visits phases (and cleanup states) in a non-deterministic order. While, the system **HS** can be solve symbolically, yielding exact closed form solutions, it may be more practical to obtain exact numeric solutions; various techniques can be used to circumvent badly conditioned matrices.

With the limiting probabilities determined in a convenient form, we compute the values of interest—$\mathbb{E}[N] = \sum_{j=0}^{\infty} j \cdot \mathbb{P}(N = j)$, $\chi = \lambda(1 - \pi_{\text{clean}}) - \eta$, π_{bad}, and π_{worse}—in terms of π_6, π_7, π_8, r_k, and $a_{m,k}$. Recalling that η is the rate at which jobs are discarded, letting $X(m)$ be the rate of initiating a cleanup event in Phase m, (i.e., $X(0) = 0$, $X(1) = X(2) = \xi_1$, $X(3) = X(4) = \xi_2$, and $X(5) = \xi_2 + \alpha_{\text{dead}}$), and observing that Phases 2 and 4 make up the **bad** malware state, while Phase 5 makes up the **worse** state, we have

$$\mathbb{E}[N] = \sum_{m=0}^{5} \sum_{k=0}^{m} \frac{a_{m,k} r_k}{(1 - r_k)^2},$$

$$\chi = \lambda(1 - \pi_6 - \pi_7 - \pi_8) - \sum_{m=0}^{5} \sum_{k=0}^{m} \frac{X(m) a_{m,k} r_k}{(1 - r_k)^2}$$

$$\pi_{\text{bad}} = \sum_{j=0}^{\infty} \left\{\pi_{(2,j)} + \pi_{(4,j)}\right\} = \sum_{k=0}^{2} \frac{a_{2,k}}{1 - r_k} + \sum_{k=0}^{4} \frac{a_{4,k}}{1 - r_k}, \qquad \text{and}$$

$$\pi_{\text{worse}} = \sum_{j=0}^{\infty} \pi_{(5,j)} = \sum_{k=0}^{5} \frac{a_{5,k}}{1 - r_k}.$$

Finally, we express $\mathcal{R} = q\chi - c \cdot \mathbb{E}[N] - \ell_{\text{bad}} \pi_{\text{bad}} - \ell_{\text{worse}} \pi_{\text{worse}}$ exactly under the **hdc**(ξ_1, ξ_2) policy:

$$\mathcal{R} = \lambda q(1 - \pi_6 - \pi_7 - \pi_8) - \sum_{m=0}^{5} \sum_{k=0}^{m} \frac{(qX(m) + c) a_{m,k} r_k}{(1 - r_k)^2} - \ell_{\text{bad}} \left(\sum_{k=0}^{2} \frac{a_{2,k}}{1 - r_k} + \sum_{k=0}^{4} \frac{a_{4,k}}{1 - r_k}\right) - \ell_{\text{worse}} \left(\sum_{k=0}^{5} \frac{a_{5,k}}{1 - r_k}\right).$$

With this expression, we can evaluate the **hdc**(ξ_1, ξ_2) family of cleanup policies, including many simple policies such as **c@slow**, **c@slower**, and **c@dead**.

5. Conclusion and directions for future work

The primary contributions of this paper are the presentation of a Markovian model for the evolution of malware on a customer-facing system and the evaluation of revenue rates under various cleanup policies. Our model moves beyond the techniques available from the literature on condition-based maintenance by incorporating queueing dynamics (for a discussion on how overlooking queueing dynamics can lead to suboptimal revenues, see Doroudi, 2016, Section 3.5). We find that in many cases, one should *not* clean a system at the first indication of a problem. In such cases, one should either wait for things to get worse, delay cleanup actions for some time, or wait until the queue lengths exceeds (or falls below) some threshold.

One of our key discoveries is that the best policies are not necessarily those that act only when a new phenomenon is observed, and rather, there are significant benefits to delaying a response for some time after witnessing a performance degradation event. One reason that these delays are beneficial is that by delaying a cleanup, one reduces downtime, while enjoying acceptable waiting times before convergence to a new steady-state with unacceptable waiting times. Another way that one can harness the benefits of persisting in a system before reaching unacceptably high waiting times is to make cleanup decisions *dynamically* by making use of queue length information. We find that even the simple *threshold* dynamic policies provide a substantial improvement over their static (non-dynamic) counterparts. Therefore, we believe that fur-

[16] $\alpha_0 = \alpha_{\text{bad}} + \gamma_{\text{slow}}$, $\alpha_1 = \alpha_{\text{bad}} + \gamma_{\text{slower}} + \xi_1$, $\alpha_2 = \alpha_{\text{worse}} + \gamma_{\text{slower}} + \xi_1$, $\alpha_3 = \alpha_{\text{bad}} + \xi_2$, $\alpha_4 = \alpha_{\text{worse}} + \xi_2$, $\alpha_5 = \alpha_{\text{dead}} + \xi_2$.

ther analytic investigation of such dynamic policies is a natural direction for future work in this area.

Another possible direction for future work in this area is the consideration of arbitrary Markovian correlation structures between the visible performance state and the hidden malware state. One could also study an enriched model where the arrival rate is *endogenously* determined based on the prices set by the service provider, the resulting mean response time, and the resulting fraction of time spent in malware states; both the customers and the service provider would be making decisions based on their own best interest.

We conclude with a discussion of a potential alternative approach that could be taken in future work toward developing strong cleanup policies in the case of hidden malware. While this paper uses Markovian performance analysis to evaluate a variety of policies, one could alternatively structure the same underlying problem as a partially observable Markov decision process (POMDP), which in this case can also be expressed as an optimal stopping time problem. Recall that at any given time, we are either in some state (m, j) or in one of the cleaning states. But while the number of jobs in the system $N = j$ is visible, the current joint malware-performance state is not. Instead, only the performance state is observable, hence, depending on the performance state, the malware state m can take on a number of (i.e., two or three) different values. We should be able to determine the probability distribution over the values taken on by m as a function of the duration of time (if any) spent in the **slow** and **slower** states since the last cleaning. Call such durations t_{slow} and t_{slower} and see that the state of the system can be expressed as $(t_{\text{slow}}, t_{\text{slower}}, j)$. Note that this state is always changing continuously throughout time, and hence we have formulated a non-Markovian problem. In this setting, a policy (or stopping condition) can be defined by a well-behaved set $A \subseteq \mathbb{R}^2 \times \{0, 1, 2, \ldots\}$ such that we initiate a cleanup at the first instance of time (since the previous cleanup) that $(t_{\text{slow}}, t_{\text{slower}}, j) \in A$. Hence, the objective is to find a policy (i.e., a "cleanup region" A) that maximizes the revenue rate. Given the non-Markovian nature of the state space, together with the difficulty of evaluating performance under A—which may require both the techniques presented in this paper and methods for determining the distribution of the joint malware-performance state as a function of $(t_{\text{slow}}, t_{\text{slower}})$—we anticipate that finding an exact optimal solution to this problem will be prohibitively difficult. That said, future work may yield fruitful heuristics or approximations by combining the state of the art in POMDP and optimal stopping time analysis with state space truncation. We hope that the analysis presented in this paper will prove helpful in the development of further heuristics for addressing the malware cleanup problem.

Acknowledgements

We thank the two anonymous reviewers for their comments that led to improving this paper. We also thank Alexander Wickeham. This work is supported by NSF-CMMI-1938909, NSF-CSR-1763701, NSF-XPS-1629444, and a 2020 Google Faculty Research Award.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.ejor.2020.10.036.

References

Alaswad, S., & Xiang, Y. (2017). A review on condition-based maintenance optimization models for stochastically deteriorating system. *Reliability Engineering & System Safety, 157*, 54–63.

Backman, J., & Ramström, H. (2018). *Efficient solving methods for POMDP-based threat defense environments on Bayesian attack graphs*. Chalmer's University of Technology Master's thesis..

Bao, T., Shoshitaishvili, Y., Wang, R., Kruegel, C., Vigna, G., & Brumley, D. (2017). How shall we play a game?: A game-theoretical model for cyber-warfare games. In *Proceedings of the 30th IEEE computer security foundations symposium (CSF)* (pp. 7–21). IEEE.

Bishop, M. (2012). *Computer security: Art and science*: 200. Boston, Massachusetts: Addison-Wesley.

Bridwell, L. (2004). *Computer virus prevalence survey*. ICSA Labs.

Bunks, C., McCarthy, D., & Al-Ani, T. (2000). Condition-based maintenance of machines using hidden Markov models. *Mechanical Systems and Signal Processing, 14*(4), 597–612.

Byon, E., & Ding, Y. (2010). Season-dependent condition-based maintenance for a wind turbine using a partially observed Markov decision process. *IEEE Transactions on Power Systems, 25*(4), 1823–1834.

Caceres, M. (2002). *Syscall proxying-simulating remote execution*. Core Security Technologies.

Cavusoglu, H., Cavusoglu, H., & Zhang, J. (2008). Security patch management: Share the burden or share the damage? *Management Science, 54*(4), 657–670.

Center for Strategic and International Studies (2018). Economic impact of cybercrime—No slowing down. Accessed 2020-10-20. https://www.mcafee.com/enterprise/en-us/assets/reports/restricted/rp-economic-impact-cybercrime.pdf.

Chakka, R., & Mitrani, I. (1994). Heterogeneous multiprocessor systems with breakdowns: Performance and optimal repair strategies. *Theoretical Computer Science, 125*(1), 91–109.

Darpa Cyber Grand Challenge (2016). Accessed 2019-2-28. https://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/.

Diamant, J., Hsu, W., Lin, D., & Scoredos, E. (2014). Automatic detection of vulnerability exploits. Accessed 2019-2-28. https://www.google.com/patents/US8739288.

Doroudi, S. (2016). *Stochastic analysis of maintenance and routing policies in queueing systems*. Carnegie Mellon University Ph.D. thesis..

Doroudi, S., Fralix, B., & Harchol-Balter, M. (2016). Clearing analysis on phases: Exact limiting probabilities for skip-free, unidirectional, quasi-birth-death processes. *Stochastic Systems, 6*(2), 420–458.

Drekic, S., & Grassmann, W. (2002). An eigenvalue approach to analyzing a finite source priority queueing model. *Annals of Operations Research, 112*(1–4), 139–152.

Ejaz, I., Alvarado, M., Gautam, N., Gebraeel, N., & Lawley, M. (2019). Condition-based maintenance for queues with degrading servers. *IEEE Transactions on Automation Science and Engineering, 16*(4), 1750–1762.

Gandhi, A., Doroudi, S., Harchol-Balter, M., & Scheller-Wolf, A. (2013). Exact analysis of the M/M/k/setup class of Markov chains via recursive renewal reward. In *Proceedings of the ACM SIGMETRICS international conference on measurement and modeling of computer systems: 41* (pp. 153–166). Pittsburgh, Pennsylvania: ACM.

Gandhi, A., Doroudi, S., Harchol-Balter, M., & Scheller-Wolf, A. (2014). Exact analysis of the M/M/k/setup class of Markov chains via recursive renewal reward. *Queueing Systems, 77*(2), 177–209.

Garetto, M., Gong, W., & Towsley, D. (2003). Modeling malware spreading dynamics. In *Proceedings of the twenty-second annual joint conference of the IEEE computer and communications IEEE societies, INFOCOM: 3* (pp. 1869–1879). San Francisco, California: IEEE.

Gatzlaff, K., & McCullough, K. (2010). The effect of data breaches on shareholder wealth. *Risk Management and Insurance Review, 13*(1), 61–83.

Haque, L., & Armstrong, M. (2007). A survey of the machine interference problem. *European Journal of Operational Research, 179*(2), 469–482.

Huang, Y., Arsenault, D., & Sood, A. (2006). Closing cluster attack windows through server redundancy and rotations. In *Proceedings of the sixth IEEE international symposium on cluster computing and the grid, CCGrid 06: 2* (pp. 12–pp). IEEE.

Hughes, L., & DeLone, G. (2007). Viruses, worms, and trojan horses: Serious crimes, nuisance, or both? *Social Science Computer Review, 25*(1), 78–98.

Latouche, G., & Ramaswami, V. (1999). *Introduction to matrix analytic methods in stochastic modeling*. Philadelphia, Pennsylvania: ASA-SIAM.

Lin, D., & Makis, V. (2003). Recursive filters for a partially observable system subject to random failure. *Advances in Applied Probability, 35*(1), 207–227.

Logan, P., & Logan, S. (2003). Bitten by a bug: A case study in malware infection. *Journal of Information Systems Education, 14*(3), 301.

Makis, V., & Jiang, X. (2003). Optimal replacement under partial observations. *Mathematics of Operations Research, 28*(2), 382–394.

Miehling, E., Rasouli, M., & Teneketzis, D. (2015). Optimal defense policies for partially observable spreading processes on Bayesian attack graphs. In *Proceedings of the second ACM workshop on moving target defense* (pp. 67–76). ACM.

Naderkhani, Z. F., & Makis, V. (2015). Optimal condition-based maintenance policy for a partially observable system with two sampling intervals. *The International Journal of Advanced Manufacturing Technology, 78*(5–8), 795–805.

Neuts, M. (1981). *Matrix-geometric solutions in stochastic models: An algorithmic approach*. Baltimore, Maryland: John Hopkins University Press.

Poolsappasit, N., Dewri, R., & Ray, I. (2012). Dynamic security risk management using Bayesian attack graphs. *IEEE Transactions on Dependable and Secure Computing, 9*(1), 61–74.

Tan, Y., Nguyen, H., Shen, Z., Gu, X., Venkatramani, C., & Rajan, D. (2012). Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Proceedings of the 32nd IEEE international conference on distributed computing systems* (pp. 285–294). IEEE.

Van Houdt, B., & van Leeuwaarden, J. (2011). Triangular M/G/1-Type and tree–like quasi-birth-death Markov chains. *INFORMS Journal on Computing, 23*(1), 165–171.

Wartenhorst, P. (1995). N parallel queueing systems with server breakdown and repair. *European Journal of Operational Research, 82*(2), 302–322.

Yue, W., & Çakanyıldırım, M. (2010). A cost-based analysis of intrusion detection system configuration under active or passive response. *Decision Support Systems, 50*(1), 21–31.