SANDIA REPORT

SAND2016-9712
Unlimited Release
Printed September 2016

An Exploration of Moving Target Defense Efficacy and Application

Christopher C. Lamb, Jason R. Hamlet

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2016-9712
Unlimited Release
Printed September 2016

An Exploration of Moving Target Defense Efficacy and Application

Christopher C. Lamb
Department 09526
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185
cclamb@sandia.gov

Jason R. Hamlet
Department 05627
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185
cclamb@sandia.gov

Abstract

Moving target defense (MTD) is an emerging paradigm in which system defenses dynamically mutate in order to decrease the overall system attack surface. Though the initial concept is promising, implementations have not been widely adopted. The field has been actively researched for over ten years, and has only produced a small amount of extensively adopted defenses, most notably, address space layout randomization (ASLR). This is despite the fact that there currently exist a variety of moving target implementations and proofs-of-concept. We suspect that this results from the moving target controls breaking critical system dependencies from the perspectives of users and administrators, as well as making things more difficult for attackers. As a result, the impact of the controls on overall system security is not sufficient to overcome the inconvenience imposed on legitimate system users. In this paper, we analyze a successful MTD approach. We study the control's dependency graphs, showing how we use graph theoretic and network properties to predict the effectiveness of the selected control. Then, with this framework in place, the dynamic nature of some Moving Target Defenses opens the possibility of modeling them with dynamic systems approaches, such as state space representations familiar from control and systems theory. We then use this approach to develop state space models for Moving Target Defenses, provide an analysis of their properties, and suggest approaches for using them.

Contents

1 Introduction **9**

Related Work 10

2 Dependency Graphs **13**

Use Case: Address Space Layout Randomization 15

Metrics 15

Scoring 17

Evaluation 18

Address Space Layout Randomization with Data Execution Prevention . . . 18

Overall Dependency Analysis 28

Optimization 28

Graph Analysis 29

Centrality 29

Community Detection 30

Cut finding 30

Efficiency 31

Application of Optimizations 31

Defense Discovery 33

3 State Space **35**

Approach 35

Open Loop 35

Output Feedback 38

State Feedback 42

Output Feedback Example: Application to Network Randomization Approaches 45

4 Conclusion **51**

References **52**

List of Figures

2.1 Dependency graphs for users and adversaries prior to applying any defenses. 19

2.2 Updated dependencies after applying address space layout randomization. Adversaries no longer have reliable knowledge of the location of injected code within memory, making NOP-sled type attacks the most likely. 22

2.3 After applying data execution prevention adversaries can no longer reliably execute injected code, and instead are likely to use return-oriented programming techniques. 23

2.4 After implementing DEP and applying ASLR to some executables the adversary is still likely to use return-oriented programming techniques, but will now need to build the malicious code only from modules that are not protected by ASLR. 25

2.5 After implementing DEP and protecting all modules with ASLR, the adversary must exploit a memory disclosure vulnerability to learn the base address of a module, and then dynamically construct malicious ROP code. 27

3.1 Open-loop state space representation for moving target defense 36

3.2 Output feedback state space representation for moving target defense. 39

3.3 State feedback with state observer state space representation for moving target defense 43

3.4 Network randomization dependency graph showing costs to fulfill dependencies prior to applying network randomization. The user's costs are all 0 because our user costs metrics consider percent increase in costs. 46

3.5 (a) Response for adversary dependencies $A28 = (n28, n26)$ and (b) $A25 = (n22, n13)$ and $A26 = (n22, n23)$ before and after IP randomization. Edges $A25$ and $A28$ are not directly impacted by the IP randomization. 50

List of Tables

2.1 Original adversary costs 20
2.2 Updated adversary costs after ASLR 21
2.3 Updated adversary costs after DEP 24
2.4 Updated adversary costs after DEP and a partial application of ASLR 26
2.5 Updated adversary costs after DEP and a complete application of ASLR 28
2.6 Summary of results 28
2.7 Edge Betweenness Centrality for Attacker Edges 32

Chapter 1

Introduction

Moving Target Defense (MTD) has been studied for over a decade. Forward thinking scientists and engineers saw the writing on the wall back then with regard to burgeoning cyber-crime. At that point, most malicious computer attacks were, for lack of a better term, not really malicious by today's standards. At that time, the hacking underground consisted of hobbyists and programmers who were exploring computer systems to see what they could really do, and what they could do with them. This was somewhat of a naive age, prior to the involvement of organized crime and terrorist organizations. But people could see where things were heading — as soon as criminal organizations understood the low level of risk and high profitability of cyber-crime, they would start to get involved. And they did, as did others, leading us to where we are today [4, 9]. The prevailing opinion was that existing defenses were insufficient to protect networked systems against the attackers of the day, let alone those of the future.

Some funding agencies began to wonder if changing a system when under threat to a higher level of overall security might be a fruitful path forward. Many physical security systems work in this way. The US military still uses the DEFCON [37] system, as well as the newer FPCON [36] system; it seemed sensible that this might be a useful avenue for research. It certainly seems valuable to have systems that can autonomically adjust their run state based on environmental context, and systems that can do this in response to an ongoing attack should be more secure than statically provisioned systems.

Furthermore, MTD approaches allow systems to dynamically update and adjust to their operating environment proactively or in response to perceived or detected threats or attacks. The intention is to provide legitimate system users and defenders with an advantage over attackers, since those attackers will find it more difficult to determine the current status of a system, or may not be able to rely on the system being in the same configuration when they launch their attack as it was when they performed reconnaissance, developed the attack, or otherwise performed their planning and preparation activities. Of course, MTDs can also make it more difficult for legitimate system users, defenders, and administrators to know the current configuration of their systems, which potentially creates opportunities for attackers. This environment provided by MTD approaches leads us to consider state space models for studying the dynamic interactions between attackers and defenders with MTDs at their disposal.

Related Work

Other researchers have modeled and studied cyber systems and the impact of moving target defenses on them. Zaffarano et. al. present a simulation framework that allows them to rapidly configure a variety of different systems topologies over which to apply moving target defenses. Many simulations of these systems are made, and data from these simulations is compiled into a variety of measures intended to represent the productivity of the system, the success of the activity model, and whether data was delivered unexposed and intact. The authors then use the mean of those measures to represent the performance of the simulated system [38]. Zhuang et. al. describe a formal logic for describing MTD systems. The logic itself takes into account system policies, constraints that must be met by operating systems, and the operational goals of a given system. It also describes sets of actions and configuration states, yielding a formal way to define moving target systems. They also address the key problems in MTD management, namely configuration and adaptation selection and timing problems, and describe why they are important to theoretical MTD analysis [40]. Additional work presents a way to model cyber attacks that takes into account overall system information, pre- and post-conditions, and attack processes that encompass multiple stages. The model is logically rigorous and enables the description of limited system dynamics in that time is referenced, but monotonically, rather than functionally [39]. A hyper-geometric probability distribution model has been proposed for studying the effects of various types of defenses within a computer network [12]. The author's specifically look at systems with no protections, systems with honeypots, and address shuffling under two attacker strategies. They find that deception (honeypots) is more successful than motion (shuffling), but that using both simultaneously provides the greatest increase in security posture. Prakash et. al. use FlipIt, a simple game in which players compete for control of a single resource, to analyze possible moving target strategies. They run the game many thousands of times, and find that each configuration of attacker and defender strategies has at least one equilibria, and in some cases many equilibria [24].

Many MTDs have been suggested in the literature. A survey of them has been provided by Okhravi *et al.* [26]. The attacker-defender dynamics in MTD have previously been studied using game theoretic models. Colbaugh and Glass analyze MTD strategies against adaptive adversaries under the assumption that the available defensive systems are independent so that attacks are effective against a single system [10]. Carter allows attacks to be effective against some subset of systems, and further assumes that the defender does not know which systems are vulnerable and is not able to detect exploits [6]. Van Dijk introduces the FlipIt game to model advanced persistent threats and targeted attacks in which one player has complete control of a resource, but in which it is not known which player controls the resource until a player makes a move, which comes at some cost [35]. Jones extends FlipIt to MTD by allowing defenders to "morph" the system to disrupt attacker knowledge [22]. Prakesh modifies FlipIt by allowing attackers to detect when control of the resource is gained by the defender, and to allow multiple target resources with objectives over the number controlled [29]. Miehling studies MTDs using partially observable Markov decision processes [25].

State space representations have been used to study both linear and nonlinear systems for many decades. They have found particular success in control engineering, where they are used to model

physical systems by inputs, outputs, and state variables, and provide a convenient means of studying the time-evolution of such systems. State space models are useful in control engineering for understanding the influence of inputs and disturbances on system state even though the models are simplified approximations of the underlying physical systems.

Chapter 2

Dependency Graphs

Despite the amount of research into moving target defense, we find that relatively few moving target approaches have been embraced in practice. It seems that this may be because the inter-dependencies and side-effects introduced by the moving target defenses were not adequately considered when the defenses were designed, and that they were not revealed during limited laboratory experiments. Furthermore, in operational settings some dynamic defenses impact users and defenders similarly to attackers, making it difficult to maintain and operate the resulting systems. We propose to study this problem by analyzing the dependencies of users, defenders, and adversaries, providing a means of determining where to locate moving target defenses, and helping to direct future research into approaches that are more likely to be transitioned to practice.

Dependencies pervade computing systems. Consider, for instance, the popular OSI model for communication systems, which has applications at the top layer, followed by the presentation, session, transport, network, data-link, and physical layers [33]. In this model higher levels are dependent on lower levels, so network layer devices have dependencies on the data-link and physical layers [11]. If the functionality of any single layer is broken, then all of the layers above it are impacted. For example, if a router stops functioning then computers connected to that router will not be able to access the Internet. Still considering the OSI model, the protocols at each level are also dependent on the protocols at lower layers. So, if an IP address (network layer) changes unexpectedly, then TCP sessions will be impacted.

It is known that the underlying MTD must have sufficient understanding of the functional and security requirements of the system, though this has not been placed in the context of dependencies [41]. Carvalho et al. have proposed a command and control system to appropriately coordinate movements in the system [7]. Our work is in identifying system details that would be needed by such a control system. Armed with this knowledge, we can then locate adversary dependencies that are not shared by users or defenders, allowing us to disrupt malicious behavior without having undue impact on legitimate users or administrators.

We begin by identifying the dependencies of the users, defenders (administrators), and adversaries of a system. By representing the dependencies of each agent on a labeled graph we can determine the overall cost of satisfying each agent's dependencies. Then, given a set of defender options, we can analyze the impact of any subset of defenses on each agent, allowing us to find that subset of defenses that will minimally disrupt users and defenders while maximally impacting attackers. If no such defenses can be found, the analysis may instead suggest new defenses.

Formally, system components and their dependencies can be represented by a labeled, directed graph, $G = (V, E, W)$, where V is a set of vertices or nodes, E is a set of ordered pairs defining directed edges between the vertices, and W is a set of weights for the edges. An edge $e \in E, e = (\mu, \nu)$ is directed from node $\mu \in V$ to node $\nu \in V$ and, in our formulation, indicates that reaching ν is dependent on first reaching μ and then satisfying the dependency that labels (μ, ν). This means that success of ν requires success of μ. An element $w \in W$ is a function of cost metrics associated with an edge. From an agents' perspective, increasing cost is detrimental. So, for instance, administrators and users want to increase the attacker's cost while minimizing their own, while the adversary desires the opposite. Potential metrics include implementation costs (time, money), memory costs (increased storage requirements), performance costs (time), network and communication costs (latency, throughput, reliability), and usability costs (support requests, system crashes). Additional metrics suggested for cyber security include time to compromise confidentiality, integrity, and availability [5, 13], defense coverage, unpredictability, and timeliness [19], and measures of deterrence, deception, and detectability [21].

We develop graphs for identified stakeholders by establishing three dependency graphs $G_u = (V_u, E_u, W_u)$, $G_d = (V_d, E_d, W_d)$, $G_a = (V_a, E_a, W_a)$, for our system to represent the dependencies of system users, system defenders, and adversaries, respectively. Note that, while we include only these three sets of agents, additional stakeholders can be added to the analysis with definitions similar to those for these three agents. The weights $w_k^u, k \in E_u$ in the user's graph are defined by $f_w^u : \mathbb{R} \to \mathbb{R}, w_k^u = f_w^u(c_i^u(k), i = 1 \ldots n_u)$ where the $c_i^u(k), i = 1 \ldots n_u$ are the n_a user costs associated with the edge $k \in E_u$. We similarly define defender labels $w_k^d, k \in E_d$ by $f_w^d : \mathbb{R} \to \mathbb{R}, w_k^d = f_w^d\left(c_i^d(k), i = 1 \ldots n_d\right)$ for defender costs $c_i^d, i = 1 \ldots n_d$ and adversary labels $w_k^a, k \in E_a$ by $f_w^a : \mathbb{R} \to \mathbb{R}, w_k^a = f_w^a(c_i^a(k), i = 1 \ldots n_a)$ for adversary costs $c_i^a, i = 1 \ldots n_a$. Note that $f_w^u(\cdot)$, $f_w^d(\cdot)$ and $f_w^a(\cdot)$ are not necessarily the same and that the definitions of the cost metrics and the number of cost metrics n_u, n_d, and n_a for users, defenders, and adversaries can also be different.

Now, we must establish the cost for satisfying the dependencies for each agent. To do so, we must find the lowest cost path to each terminal node from a root node. We define the cost of a path as a function of the costs associated with the edges along that path. Letting $w_i^u, i = 1 \ldots n$ be the set of weights along a path p_i^u from a root node to a terminal node in the user's graph, we define the cost of the path as $f_c^u : \mathbb{R} \to \mathbb{R}, p_i^u = f_c^u(w_i^u, i = 1 \ldots n)$. Given this definition, root node $\alpha \in V_u$, and terminal node $\zeta \in V_u$, Dijkstra provides an algorithm for finding the lowest cost path from α to ζ [15]. We similarly define the cost of a path p_i^d in the defenders' graph as $f_c^d : \mathbb{R} \to \mathbb{R}, p_j^d = f_c^d\left(w_i^d, i = 1 \ldots n\right)$ and the cost of a path p_j^a in the adversary's graph by $f_c^a : \mathbb{R} \to \mathbb{R}, p_j^a = f_c^a(w_i^a, i = 1 \ldots n)$ and, of course, can use the same algorithm for finding the lowest cost paths.

Now, we require a method for determining the overall cost for each agent. To do so, let $P_u = \{p_{i^u, i=m_u}\}, P_d = \left\{p_{i^d, i=m_d}\right\}$ and $P_a = \{p_{i^a, i=m_a}\}$ be the sets of the lowest cost paths satisfying all m_u, m_d and m_a of the user's, defender's, and adversary's terminal dependencies, respectively. Then define the user's overall cost, s_u, by $f_s^u : \mathbb{R} \to \mathbb{R}, s_u = f_s^u(p_i^u, i = 1 \ldots m_u)$. Likewise, the defender and adversary costs are defined by $f_s^d : \mathbb{R} \to \mathbb{R}, s_d = f_s^d(p_i^d, i = 1 \ldots m_d)$ and $f_s^a : \mathbb{R} \to \mathbb{R}, s_a = f_s^a(p_i^a, i = 1 \ldots m_a)$, respectively.

Use Case: Address Space Layout Randomization

As an example of this dependency graph based approach, we now consider applying it to address space layout randomization (ASLR). ASLR is a code location randomization technique used to help defeat Return Oriented Programming attacks, or other types of attacks that depend on the attacker having some a priori knowledge of the code map of a running program. By randomizing the locations of various libraries and functions in a running program image, the level of difficulty of developing exploits increases significantly. ASLR is widely implemented in modern operating systems, appearing in Windows, Linux, OS X, Android, and iOS.

ASLR is typically used in conjunction with Data Execution Prevention (DEP). DEP marks areas in memory as being *executable* or *non-executable*. Only code appearing in the executable regions can be run by programs. Hardware-enforced DEP, which uses hardware to mark pages as executable or non-executable, is most effective, but software-enforced DEP is also available. DEP protects against attacks that rely on executing instructions located in non-executable pages. It is especially useful against buffer overflows, since these attacks often store instructions in non-executable memory locations. Like ASLR, DEP is also widely deployed and appears in Windows, Linux, OS X, iOS, and Android operating systems.

DEP was originally developed to stop the proliferation of stack-overflow based attacks, through which attackers could easily inject shellcode into running programs via taking advantage of the stored instruction pointer. Typically, attackers would overwrite a buffer such that the memory after the buffer would be tactically overwritten to insert a new value into the stored instruction pointer. This causes the program to resume execution at an overwritten location below the instruction pointer at which the attacker was able to inject valid code. DEP prevented this attack by prohibiting execution from these memory regions. This lead to the development of Return-Oriented Programming, a programming paradigm in which attackers jump to single instructions (or parts of instructions) within the running program. With a static, known runtime program image, these instruction locations are easily reusable, once located. The solution to this problem was to randomize the layout of the address space, which moves these instructions so that they cannot be predicted based on previous experimentation.

Individually, DEP and ASLR are not particularly effective. Attackers can compromise systems without DEP via stack overflow techniques, and DEP without layout randomization is compromised via ROP. When combined however, they are particularly effective. For this use case, we consider only defenders and attackers.

Metrics

We use different metrics for evaluating the costs of users and adversaries fulfilling their respective dependencies. For adversaries our metrics are:

- time to acquire access

- cost to acquire access
- time to acquire knowledge
- cost to acquire knowledge
- unpredictability
- frequency of movement

The first two metrics represent the adversary's difficulty in achieving the proper position to complete an action or to fulfill a dependency. For example, an adversary wishing to exploit a buffer overflow must first be in a position to write to the buffer. The next two metrics describe the skills and knowledge required by an adversary to be successful in the attack. The final two metrics describe the uncertainty that an attacker will face when attempting to complete an attack step. We estimate this uncertainty by how predictable the operating environment is to the adversary and by how often this environment changes. Additional metrics, such as the size of the attack team and their commitment to the attack could also be employed [16]. We found that, for this example, additional metrics did not help to differentiate the attack steps. For each edge in the adversary's dependency graph we evaluate each of these metrics on a "low-medium-high" scale.

For users our metrics are:

- change in memory requirements
- change in CPU requirements
- change in system stability
- change in networked communication latency
- change in networked communication bandwidth
- change in networked communication stability

These metrics focus on disruption to the user's computing experience with respect to both computational and networking or communication overhead. For scoring, we initially assign the cost of fulfilling each dependency as zero. This is because the initial state is an existing system or implementation, so no overhead results from maintaining this state. Then, we evaluate the cost of a defense by estimating the percent change in each metric that will result from applying the defense.

Note that the specific set of metrics used for this analysis can be changed to suit the application or system under study. Additionally, we also note that the absolute scores assigned to each metric for each edge are less important than the relative scores between edges. This is because assigning a particular cost to an edge or path is less important than knowing which paths are the most or least expensive. Due to this, consistency in assignment of scores is necessary. For example, if a "low-medium-high" scale is used to score the attacker metrics then it is useful to define boundaries or ranges for each of the scores to aid in consistent application of them.

Scoring

Users and adversaries alike have sets of dependencies that they must satisfy in order to achieve their goals. These sets are represented by paths from the initial node to a terminal node in the dependency graph. The cost of completing the goal is then the cost of satisfying each of the dependencies. Consequently, we need to estimate the aggregate cost of fulfilling a set of dependencies.

It does not make sense to add the individual costs since this unfairly penalizes longer paths, which may occur simply because some portions of the dependency graph are more detailed than others. We also know that it probably does not make sense to use the maximum individual cost as the composite cost since, for instance, multi-stage attacks with several equally and highly expensive steps are likely more costly for attackers than attacks with only one such difficult step.

We transform the cost of each stage of a dependency path into a value between 0 and 1. For this, we map more costly steps to values closer to 0, and easier steps to values close to 1. Conceptually, we think of these values as representing the probability of success, although this interpretation should not be taken literally. Now, after transforming the costs, and using our probabilistic interpretation, we can compose them by finding the joint probability of success of the path. The joint probability of success is strictly smaller than any of the individual probabilities of success. This means that the joint probability will include contributions from each individual dependency; easily satisfied dependencies are not assumed to be fulfilled, although they will have less impact than more difficult steps. After finding the joint probability, we can then transform it back into a cost by inverting the original transformation.

The transformation we propose is

$$p = \frac{m-d}{m} \tag{2.1}$$

where m is an upper bound for the cost of a step, d is the cost of a step, and p is the value that we interpret as the probability of success for this step. To aggregate the probabilities of success across a set of edges $e_i, i = 1 \ldots n$ we simply calculate $\prod_i p_i$. The inverse transformation corresponding to equation 2.1 is

$$d = m - pm \tag{2.2}$$

and so the composed cost of the multi-stage dependency is

$$c = m - m\prod_i p_i \tag{2.3}$$

where the v_i correspond to the individual steps in the dependency path.

Evaluation

We are generally interested in answering questions about the costs of paths through the dependency graph from the start node to an ending node. Typically, we are interested in finding the adversary's least costly paths for satisfying all of the dependencies for a particular attack, and in finding the most costly paths for users, administrators, and other legitimate users. This is because we would like for our defenses to maximize the cost of the adversary's least expensive paths, while minimizing the cost of legitimate actors' most expensive paths. Djikstra's algorithm is suited to solving this problem if we use eqn. 2.3 as the distance metric [15].

Address Space Layout Randomization with Data Execution Prevention

In our dependency graphs each directed edge is labeled with three values. The first label indicates whether the edge is an adversary or user dependency, and is of the form aN or uN, where N is an integer and a and u indicates adversary or user, respectively. The next label describes the dependency, and the third label is the cost for satisfying the dependency. For visual reference, adversary edges and nodes are depicted in orange, and user edges and nodes in purple. Shared nodes are shown in white. Edges that are made prohibitively expensive by a defensive action are denoted by gray dashed lines, and nodes that are cutoff from the graph by defensive actions are also shown in gray.

We consider a running example in which a defender attempts to prevent attacks by implementing ASLR and DEP. We first consider the initial system state before any defense is deployed. We then consider ASLR and DEP independently, DEP combined with a partial implementation of ASLR, and DEP combined with a complete implementation of ASLR. In each case we examine likely attacker approaches responding to the defense.

First, we discuss the initial scenario before any defenses are deployed. Considering Figure 2.1, we first describe why this particular set of dependencies was selected. In this case, we include user dependencies that capture local code execution. This involves executing subroutines and other code that is located at different locations in the address space since not all instructions appear at sequential addresses. Host application attacks typically require jumping to or otherwise redirecting the execution flow to injected code. This can be accomplished by placing the code at an address that appears on the stack as a return address, so we must include attacker dependencies for accomplishing this. The users' dependency graph is presented in the purple path in Figure 2.1, where we see that the users' dependencies include calling the subroutine, pushing the return address onto the stack, executing the subroutine, and writing data into a buffer. After the subroutine is complete the return address is popped off the stack, the instruction pointer is updated, and the original program execution continues. Since this is the initial system configuration and the user's costs are defined as changes from some baseline, all of user's dependencies are assigned zero cost. The adversary's original dependencies are shown by the orange paths in Figure 2.1. In this initial setting, the adversary can succeed at injecting and executing malicious code by exploiting a simple buffer overflow vulnerability. For this, the adversary must have knowledge of the machine architecture, locate a vulnerable buffer, and be able to write data into this buffer. By overflowing this

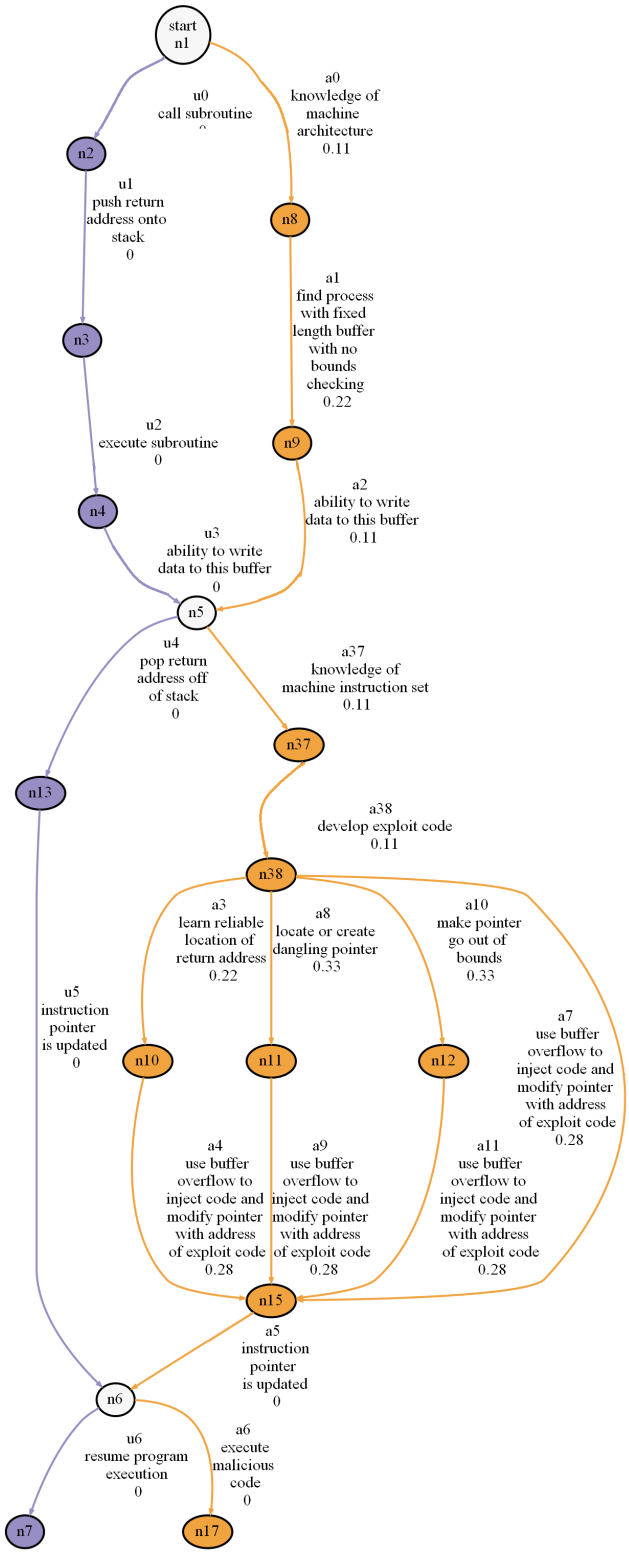

Figure 2.1. Dependency graphs for users and adversaries prior to applying any defenses.

buffer, the adversary is able to inject the malicious code. Note that, owing to the monoculture of defense postures, the adversary can identify the vulnerable process and write the buffer overflow exploit for it offline. Now, the adversary uses any of several methods for overflowing the buffer and redirecting program execution to the malicious code. Approaches for this may include overwriting the return address on the stack, or making use of a dangling or out of bounds pointer. In any case, if the adversary is able overflow the buffer to inject the malicious code and then update the instruction pointer to point to this code, then the malicious program will be executed. Due to the static, predictable defense posture presented to the adversary in this scenario, we find that the cost for satisfying the adversary's dependencies is low in this initial setting. We score each of the attacker's metrics on a scale from $0-1$, with 1 being the most costly. For this example, we only allow scores of $0, 1/3, 2/3$ and 1, although finer gradations are possible. The initial attacker costs are presented in Table 2.1. In Table 2.1 t_a is the attacker's time to acquire access, c_a is the cost to acquire access, t_k and c_k are the time and cost to acquire knowledge, u is unpredicitability, f_m is the frequency of movement, and o_c is the overall cost.

Table 2.1. Original adversary costs

	t_a	c_a	t_k	c_k	u	f_m	o_c
a0	0.00	0.00	0.33	0.33	0.00	0.00	0.11
a1	0.00	0.00	0.67	0.67	0.00	0.00	0.22
a2	0.33	0.33	0.00	0.00	0.00	0.00	0.11
a3	0.00	0.00	0.67	0.67	0.00	0.00	0.22
a4	0.33	0.33	0.67	0.33	0.00	0.00	0.28
a5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a6	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a7	0.33	0.33	0.67	0.33	0.00	0.00	0.28
a8	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a9	0.33	0.33	0.67	0.33	0.00	0.00	0.28
a10	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a11	0.33	0.33	0.67	0.33	0.00	0.00	0.28
a37	0.00	0.00	0.33	0.33	0.00	0.00	0.11
a38	0.00	0.00	0.33	0.33	0.00	0.00	0.11

Figure 2.2 shows updated dependencies after applying ASLR. Note that the cost of ASLR to the user is essentially zero, with only a slight increase in CPU overhead for randomizing the memory space when a process is initialized. This change is shown in Figure 2.2, where we have included a 10% change in CPU requirements completing dependency $u0$. This manifests as a total increase in cost of less than 1%, which is consistent with the literature on 32-bit ASLR [34]. However, the randomized memory layout has a greater impact on the cost of fulfilling the adversary's dependencies since key elements, such as the location of the stack and buffers and the base address of loaded modules are now randomized. These increased costs are also shown in Figure 2.2. Some of the adversary dependencies, such as learning the reliable location of the return address and modifying a pointer with the address of exploit code, have more costly access requirements due to the effort required for learning the appropriate return addresses. Some of the dependencies also now require a more sophisticated adversary, increasing the cost of acquiring the knowledge required for a successful attack. The greatest increases in cost, however, arise from the unpredictability introduced to the system by ASLR. Since the adversary can no longer identify

reliable addresses before launching an attack, the adversary now faces a less predictable operating environment. However, it is still possible for the adversary to be successful. We find that the cost of fulfilling the adversary's dependencies for a successful attack increases by 29%. The updated costs are presented in Table 2.2. One option available to the adversary is to brute force or guess the locations of the required structures. While this may be possible in some circumstances, it is also risky and may lead to detection. We find that the more likely attack is to employ a NOP-sled or other heap-spraying technique, which is consistent with the literature [30]. In these approaches, a large section of memory is filled with NOP instructions, with the malicious code following the NOPs. Jumping anywhere within the sequence of NOPs will eventually cause the malicious code to execute. We add this as a potential adversary dependency as well.

Table 2.2. Updated adversary costs after ASLR

	t_a	c_a	t_k	c_k	u	f_m	o_c
a3	0.00	0.00	0.67	0.67	0.00	0.00	0.22
a6	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a22	0.00	0.00	0.00	0.00	0.33	0.33	0.11

In Figure 2.3 we present dependency graphs showing the influence of DEP. The cost to users is low, with only minor computational overhead and a slight decrease in system stability. The decreased stability is primarily caused by legacy applications that do not conform to the memory restrictions enforced by DEP. We estimate these costs as a 5% increase in CPU requirements for edges $u1, u3$ and $u6$ and 5% decrease in system stability for $u2$. We estimate the total increased cost to users at less than 1%. There is a much larger increase in the cost of fulfilling the adversary's dependencies. Since code injected with a buffer overflow is likely to be in memory marked as non-executable, adversaries must find new methods of attack. Typically, these attacks use return-oriented programming [28]. This technique strings together sequences of instructions from existing programs. Since these instructions are in memory locations marked as executable, the adversary need only to locate an acceptable sequence and then update the program counter to follow the desired sequence. Since DEP does not introduce any randomization, ROP exploits can be identified and written offline. We represent this by adding attacker dependencies for identifying the ROP code at the top of Figure 2.3. The attacker must still inject some exploit code to hijack control of execution, likely with a buffer overflow attack to overwrite the return address, and so this set of dependencies still appears. After this, there are several approaches the attacker may take for executing the exploit that depend on the particular construction of the exploit code. These appear at the bottom of Figure 2.3. Constructing ROP attacks is more difficult than writing standard shellcode exploits, and consequently requires more knowledgeable adversaries. Additionally, execution of these exploits requires not only construction of the ROP code, but also modification of the instruction pointer. In Figure 2.3 these distinct dependencies are indicated by labeling sets of adversary terminal nodes as '(0)' and '(1)'. The adversary must fulfill dependencies from both sets in order to be successful. Due to the additional requirements on adversary capability for developing the attack and the distinct dependency paths for development and injection of the attack, we find that DEP increases the adversary's cost by about 52%, which is larger than with ASLR. Changes from the attacker's original costs appear in Table 2.3

Individually, both ASLR and DEP can be bypassed relatively easily with well-known tech-

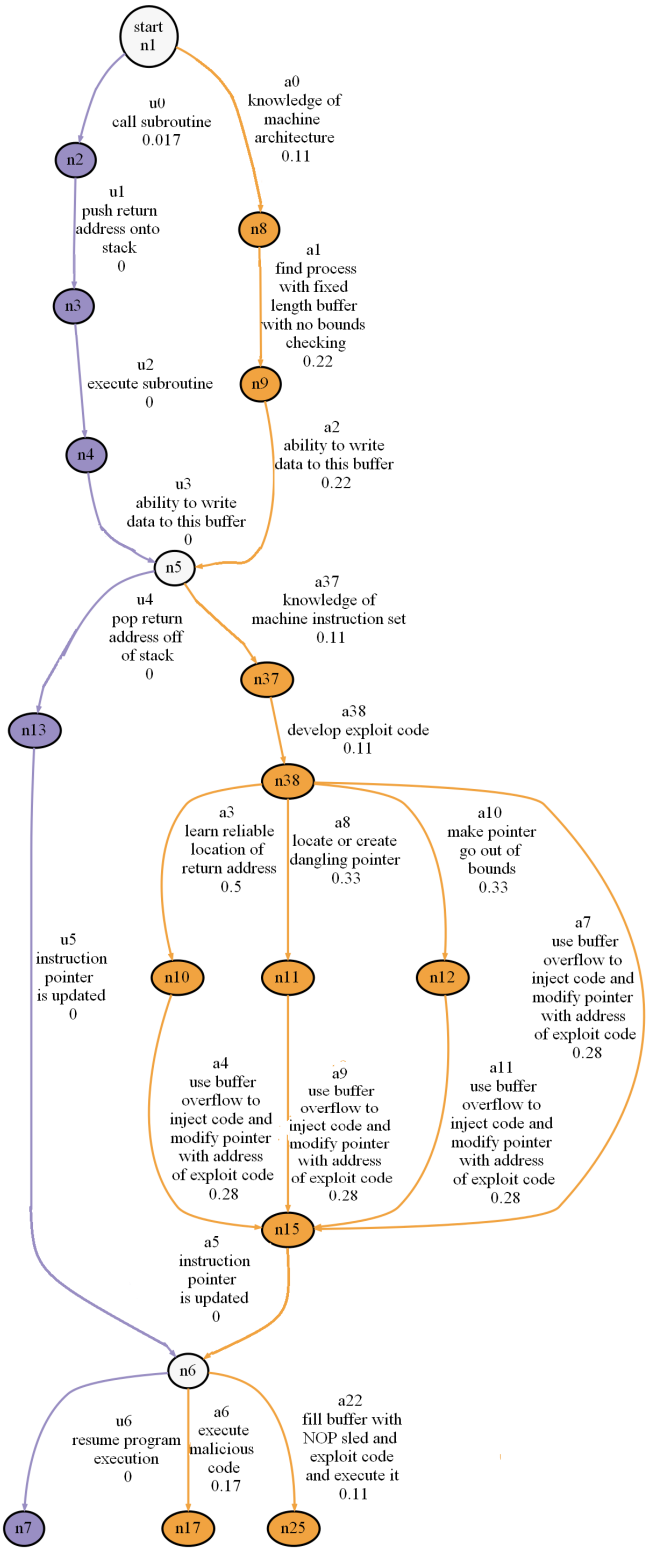

Figure 2.2. Updated dependencies after applying address space layout randomization. Adversaries no longer have reliable knowledge of the location of injected code within memory, making NOP-sled type attacks the most likely.

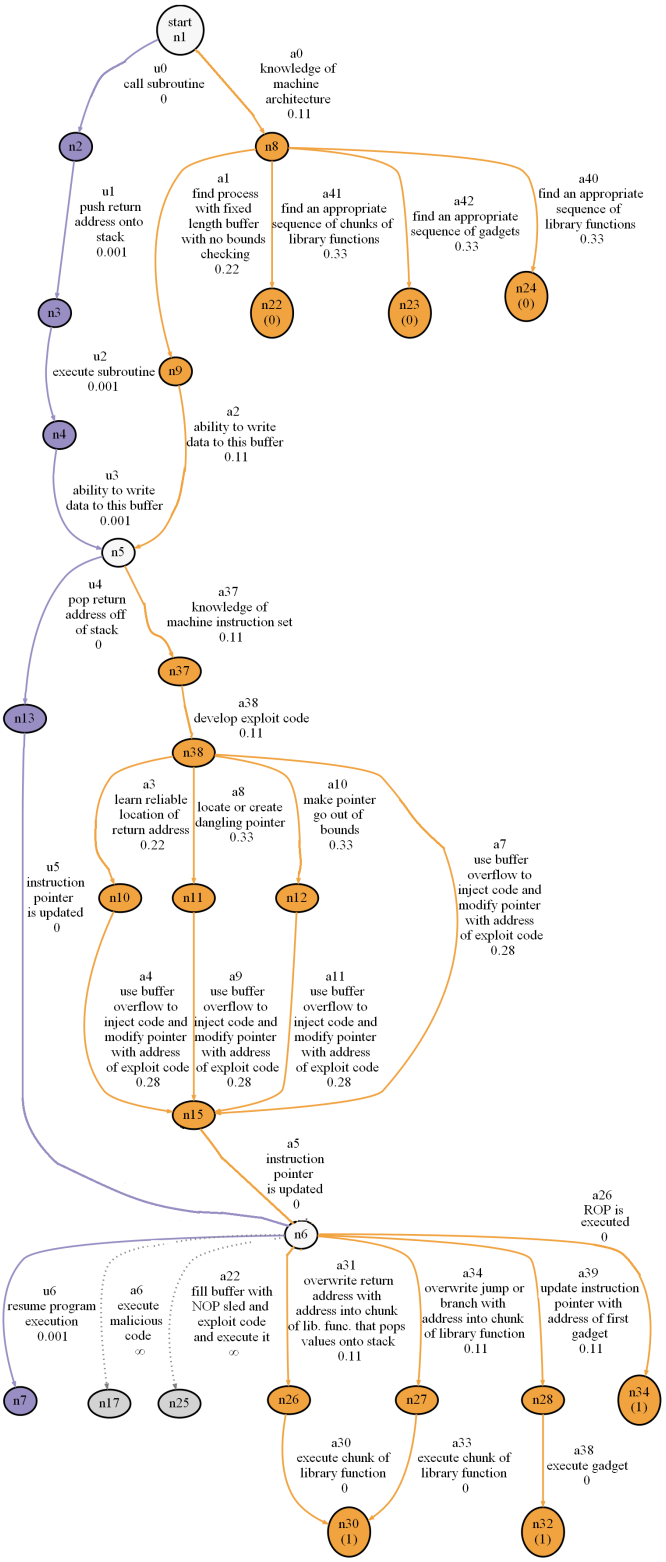

Figure 2.3. After applying data execution prevention adversaries can no longer reliably execute injected code, and instead are likely to use return-oriented programming techniques.

Table 2.3. Updated adversary costs after DEP

	t_a	c_a	t_k	c_k	u	f_m	o_c
a6	1.00	1.00	1.00	1.00	1.00	1.00	1.00
a22	1.00	1.00	1.00	1.00	1.00	1.00	1.00
a26	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a30	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a31	0.00	0.00	0.00	0.00	0.33	0.33	0.11
a33	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a34	0.00	0.00	0.00	0.00	0.33	0.33	0.11
a38	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a39	0.00	0.00	0.00	0.00	0.33	0.33	0.11
a40	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a41	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a42	0.33	0.33	0.67	0.67	0.00	0.00	0.33

niques, and so they are not particularly effective when used independently [34, 32]. However, they are complementary techniques and can be employed together. One method for doing this is to use DEP in conjunction with a limited application of ASLR in which not all modules are protected. Dependency graphs for this are shown in Figure 2.4. Here, we find that because of DEP the adversary is still likely to use a ROP attack. However, ASLR prevents the adversary from knowing memory locations for all executables in advance, so the adversary must locate modules that are not protected by ASLR and then build the ROP program from those modules. Consequently, we add an attacker dependency of finding an executable or relative address that is not protected by ASLR from which to build the ROP attack. This limitation on the adversary's access complicates the attack and requires circumventing both the ASLR and DEP protections. Since these protections are dissimilar, defeating one does not imply the ability to defeat the other. This effective independence increases the adversary's cost considerably. We find a 117% increase in adversary cost, which is slightly more than the combined increase required for defeating both ASLR and DEP independently. The changes from the attacker's original costs and assessment of new edges appears in Table 2.4. As before, we find that the user's costs increase by less than 1%. This represents a 5% increase in CPU requirements for edges $u1, u3$ and $u6$, a 10% increase in CPU requirements for $u0$ and 5% decrease in system stability for $u2$.

We can also pair DEP with a complete application of ASLR that protects all modules. In the scenario, the adversary can no longer rely on any code being at known locations. The adversary must now learn the location of a module using some memory disclosure vulnerability, and then exploit this vulnerability to dynamically construct a ROP payload. Following [14], we represent this by incorporating attacker dependencies for using a memory disclosure vulnerability to find the base address of a dynamically linked library, and then using this to dynamically construct the ROP payload. While the basic process of using ROP to bypass DEP is unchanged, the skills and difficulty of constructing the exploit increase greatly, and in Figure 2.5 we find a 168% increase in the adversary's cost. Changes from the attacker's DEP and partial ASLR costs and assessment of new dependencies appear in Table 2.5. Note that we have assigned the maximum possible cost to $a15$ and $a17$, indicating that those edges have been removed by the defensive action. As before, users are essentially unaffected by the protections, and their costs increase by less than 1%. As

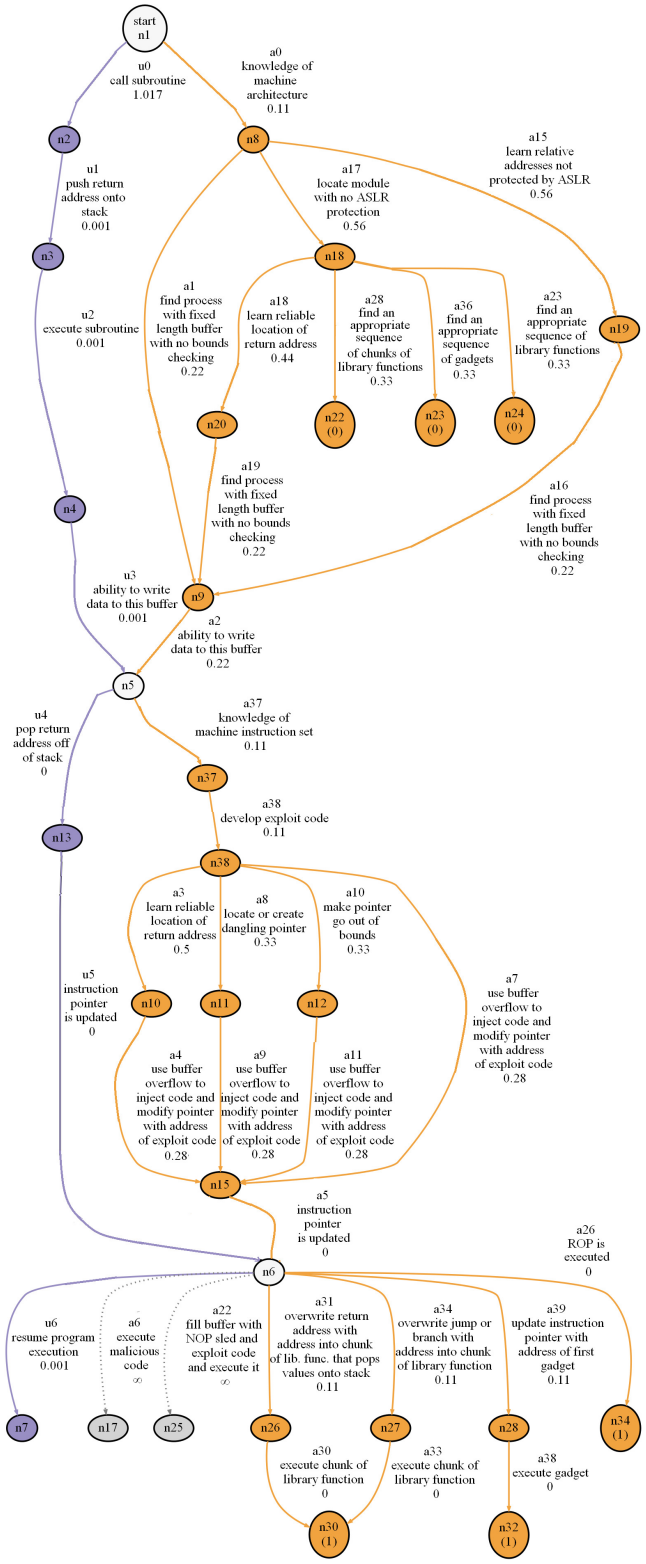

Figure 2.4. After implementing DEP and applying ASLR to some executables the adversary is still likely to use return-oriented programming techniques, but will now need to build the malicious code only from modules that are not protected by ASLR.

Table 2.4. Updated adversary costs after DEP and a partial application of ASLR

	t_a	c_a	t_k	c_k	u	f_m	o_c
a2	0.33	0.33	0.00	0.00	0.33	0.33	0.22
a3	0.33	0.33	0.67	1.00	0.33	0.33	0.50
a6	1.00	1.00	1.00	1.00	1.00	1.00	1.00
a15	0.33	0.33	1.00	1.00	0.33	0.33	0.56
a16	0.00	0.00	0.67	0.67	0.00	0.00	0.22
a17	0.33	0.33	1.00	1.00	0.33	0.33	0.56
a18	0.33	0.33	0.67	0.67	0.33	0.33	0.44
a19	0.00	0.00	0.67	0.67	0.00	0.00	0.22
a22	1.00	1.00	1.00	1.00	1.00	1.00	1.00
a23	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a26	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a28	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a30	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a31	0.00	0.00	0.00	0.00	0.33	0.33	0.11
a33	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a34	0.00	0.00	0.00	0.00	0.33	0.33	0.11
a36	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a38	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a39	0.00	0.00	0.00	0.00	0.33	0.33	0.11

with DEP and partial ASLR, these costs are incurred through a 5% increase in CPU requirements for edges $u1, u3$ and $u6$, a 10% increase in CPU requirements for $u0$ and 5% decrease in system stability for $u2$. The results from our analysis of these four scenarios are summarized in Table 2.6.

Finally, we also note that it is possible for an attacker to bypass a protection entirely. For example, an attacker might bypass ASLR by targeting an application that leaks memory information and then using this to bypass ASLR, or by simply turning ASLR off. After bypassing ASLR the attacker can then proceed with a DEP resistant exploit, such as a ROP attack that uses instructions at memory locations that the attacker can learn after bypassing ASLR. Our modeling approach is intended to assess the costs incurred by the various classes of users when a particular defense is correctly deployed. Due to this, we model the costs to an attacker to complete the attack assuming that the defense is in place. Consequently, while we acknowledge the possibility of defenses being bypassed entirely, we do not attempt to capture it in our models since there is little to gain by modeling a defense and then simply bypassing it, which effectively removes it from the model. Additionally, while we present ASLR and DEP as an example of the modeling approach, we reiterate that a primary benefit of the modeling approach is for identifying attacker dependencies that are suitable for targeting with a defense. Or initial motivating use and target application may be moving target or dynamic defense, but the approach is suitable for a variety of different environments.

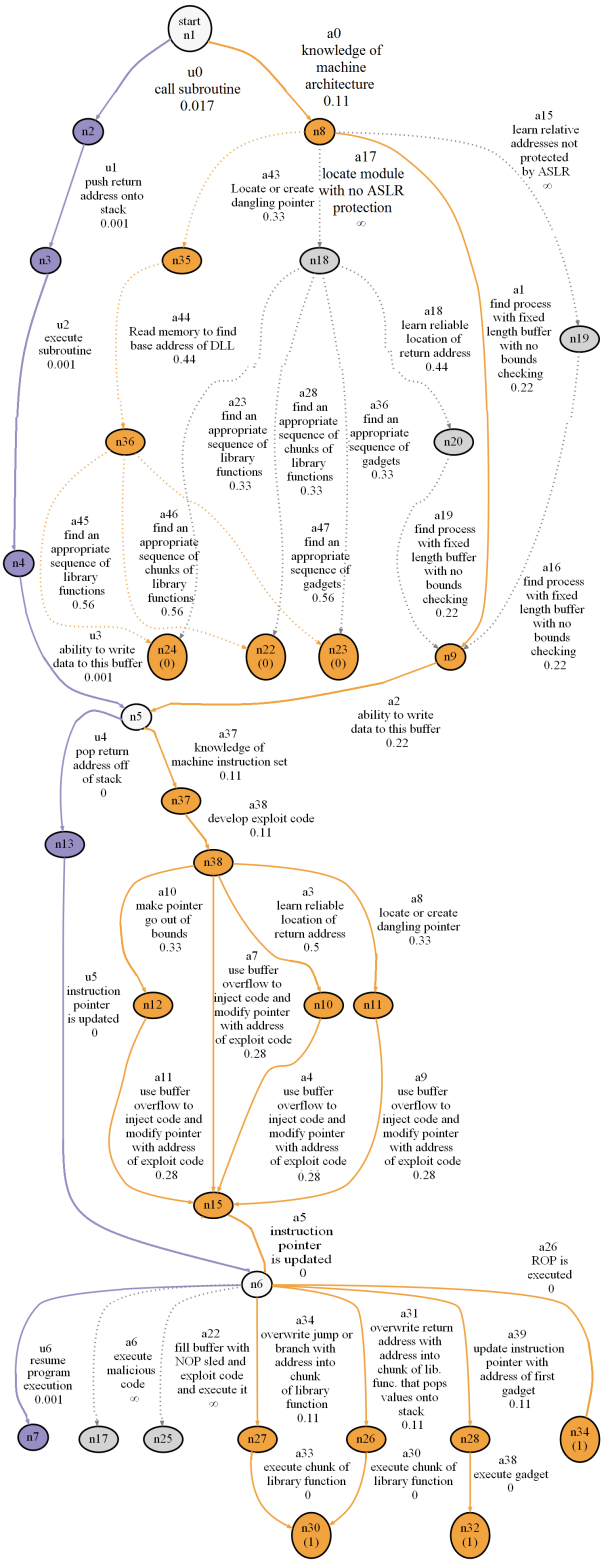

Figure 2.5. After implementing DEP and protecting all modules with ASLR, the adversary must exploit a memory disclosure vulnerability to learn the base address of a module, and then dynamically construct malicious ROP code.

Table 2.5. Updated adversary costs after DEP and a complete application of ASLR

	t_a	c_a	t_k	c_k	u	f_m	o_c
a15	1.00	1.00	1.00	1.00	1.00	1.00	1.00
a17	1.00	1.00	1.00	1.00	1.00	1.00	1.00
a19	0.00	0.00	0.67	0.67	0.00	0.00	0.22
a28	0.33	0.33	0.67	0.67	0.00	0.00	0.33
a29	0.33	0.33	0.67	0.67	0.33	0.33	0.44
a30	0.33	0.33	1.00	1.00	0.33	0.33	0.56
a31	0.33	0.33	1.00	1.00	0.33	0.33	0.56
a32	0.33	0.33	1.00	1.00	0.33	0.33	0.56

Table 2.6. Summary of results

defensive technique	**Δ adversary cost**	**Δ user cost**	**Δ adversary cost / Δ user cost**
ASLR	23%	0.25%	92
DEP	47%	0.06%	783
partial ASLR and DEP	117%	0.29%	403
full ASLR and DEP	163%	0.29%	562

Overall Dependency Analysis

Optimization

In section 2 we demonstrated the use of dependency graphs to analyze a moving target defense. We also applied Djikstra's shortest path finding algorithm with a modified distance metric, to find the overall user and attacker costs for fulfilling all of the relevant dependencies. Now, we explore additional analytic approaches for studying dependency graphs and suggest how these approaches could be applied to analysis of MTDs.

Recall from section 2 that the users', defenders', and attackers' overall costs for fulfilling their dependencies are s_u, s_d, and s_a, respectively. To identify locations most suitable for applying existing MTDs, or even to identify attacker dependencies that could be impacted by new defenses without burdening users or defenders, we want to find those ways to maximally impact the adversary while minimally impacting users and defenders. One way to do so is by solving the multi-objective optimization problem $\min(s_u, s_d, -s_a)$. We use $-s_a$ in the formulation of the optimization problem since minimizing it is equivalent to maximizing the adversary's cost. The optimization itself can be constrained by defining a list of defender options and associating with each of them the impact that they will have on the user, defender, and adversary cost metrics $c_i^u, i = 1 \ldots n_u$, $c_i^d, i = 1 \ldots n_d$, and $c_i^a, i = 1 \ldots n_a$. In general, these impacts vary from edge to edge within a graph, although in practice a particular defense option will influence only a subset of edges within a graph and so the

impact of the defense mechanism only needs to be determined for that set of edges. If no suitable solution to the multi-objective optimization problem can be found, or due to interest in discovering new defensive approaches, this analysis can be expanded to allow for defense discovery, which can be aided by graph analysis techniques.

Graph Analysis

There are many graph analysis tools and algorithms that are appropriate for analyzing dependency graphs. Algorithms and approaches for detecting or identifying bottlenecks, popular nodes, low costs paths, communities belonging solely to users, defenders, or attackers, and cuts that separate such communities, allowing them to be targeted, are all of interest. Here, we provide a brief overview of some of the relevant graph analysis techniques for achieving these goals. Formally, we define a multigraph $G_m = (V_m, E_m, W_m)$ where $V_m = V_u \cup V_d \cup V_a$, $E_m = E_u \cup E_d \cup E_a$, $E_i \cap E_j = \emptyset, i \neq j$, and $W_m = W_u \cup W_d \cup W_a$, $W_i \cap W_j = \emptyset, i \neq j$, which is simply a single graph produced by combing the user, attacker, and defender digraphs.

Centrality

Graph centrality measures attempt to identify the most important nodes or edges within a graph. We consider both edge betweenness centrality and eigenvector centrality. The betweenness centrality of an edge e is the sum of the fraction of all pairs shortest paths that pass through e [3]. It is calculated as $c(v) = \sum_{s,t \in V} \frac{\sigma(s,t|e)}{\sigma(s,t)}$ where V is the set of nodes in the graph, e is the edge under consideration, $\sigma(s,t)$ is the number of shortest paths from node s to node t, and $\sigma(s,t|e)$ is the number of those shortest paths that pass through e.Using this definition of centrality, increasing adversary costs on the in-edges of central nodes is desirable since doing so will increase the adversary's costs for satisfying even the least expensive dependencies. If enough such defenses are available, the adversary's costs can all be increased to some minimum value. This may have the effect of eliminating some attackers from the system entirely. Considering the user's and defender's dependencies, defensive maneuvers that impact nodes with large betweenness centrality scores is desirable because such defenses will impact shortest-paths in the dependency graph. Increasing the cost of these shortest paths is more desirable than increasing the cost of paths that are already expensive to satisfy.

With eigenvector centrality connections to high-scoring nodes contribute more to a node's score than connections to lower scoring nodes. This is appealing for our study of dependency graphs since it permits us to identify not only the most central nodes, but also the nodes that lead to these central nodes. This allows us to consider defenses that impact not only the most central nodes, but also to identify and impact nodes that are connected to them. This broadens the number of edges available for targeting while also increasing the adversary's costs for satisfying common dependencies. If we can find defenses that impact paths leading to user and defender nodes that have low centrality scores then this might limit the impact on users and defenders since these paths are not exercised very frequently.

Dissimilarity centrality measures assign more relevance to nodes with greater dissimilarity, since those nodes allow the given node access to portions of the graph that the given node cannot access directly. For example, if there are two clusters of nodes in a graph and these clusters are connected by a single edge, then the nodes connected by this edge are more central (most dissimilar) because they permit access to the different clusters. Targeting these edges in the attacker's graph allows us to increase the cost of traversing bottlenecks in the attacker's graph. Conversely, finding such bottlenecks in user's or defender's graphs may help us to identify defenses or system changes to add parallel paths or to increase the connectivity of the legitimate actor's graphs.

Community Detection

Graphs have community structure if their nodes can be grouped into subsets that are internally densely connected. Such communities are of interest to us for a variety of reasons. If we identify communities within attacker graphs, then we may be able to target defenses to increase the costs of satisfying edges within a community, or we may be able to find central or influential nodes within individual communities to target with our defenses. Similarly, we may also be able to identify communities in the defender graphs and then seek defensive measures such as increasing the number of communities or reducing the centrality of nodes within communities to limit the negative impacts of attacker and defender actions on the defenders. Additionally, we may attempt to divide the defender and attacker nodes into distinct communities to limit the negative impact of defender actions on the defender.

Community detection is useful for defense discovery. It allows us to look for adversary dependencies that are not shared by users or defenders. These are the edges $e = (\mu, \nu) \in E_a$ s.t. $\mu, \nu \notin V_u \cup V_d$. Any such edges are dependencies belonging only to the adversary, and so defensive measures targeting those edges, and no others, will impact only adversarial operations.

The Girvan-Newman algorithm is one approach for community detection [17]. The Girvan-Newman algorithm detects edges that are most likely between communities by finding those edges that appear along many shortest paths. These edges are then removed from the graph, and the process repeats. In a network with community structure containing two or more internally densely connected communities, but with few edges connecting them, those edges that connect communities will have high edge-betweenness and are targeted for removal. Eventually, only the densely connected communities remain in the graph. This approach is useful for finding central nodes in networks that have known starting and ending points, which is true for dependency graphs.

Cut finding

The minimum cut of a graph is a bottleneck in the graph. We want to find these bottlenecks and either remove them for defenders or attempt to create or strengthen them for attackers. Additionally, finding cuts of the graph that separate or almost separate it into defender and attacker graphs may allow us to identify defenses that preferentially impact the attacker's edges more so than the defender's edges. Similarly, such cuts may also suggest locations for adding

new defender or attacker requirements to either make it easier for the defender to preferentially target attacker dependencies, or to make it more difficult for the attacker to preferentially target defender dependencies. For instance, we can search for cuts in the merged graph that disconnect more of the adversary's nodes than user and defender nodes. If the user and defender nodes can be reconnected to their original graphs, for example, by adding additional vertices, then these cuts will identify adversary edges that can be targeted for new defensive measures and which will require minimal additional edges to be added to the user and defender graphs. In particular, if we can find a cut $C = (S,T) = \{(s,t) \in E_m | s \in S, t \in T\}$ of $G_m = (V_m, E_m, W_m)$ s.t. $|\{t \in T \cap E_u\}| \leq |\{t \in T \cap E_d\}| < |\{t \in T \cap E_a\}|$ then we might consider defender actions that will impact the edges in T. In practice, it is possible that these defender actions will not actually partition G_m, but rather that they will increase the weights associated with the edges in T. Although these actions will also impact users and defenders, if the weights in W_u and W_d associated with the edges $\{t \in T \cap E_u\}$ and $\{t \in T \cap E_d\}$ do not increase too much, or if additional edges e_u and e_d that are not impacted by the defender action can be added to E_u and E_d, then these edges effectively patch the user and defender systems, reducing the impact of the defensive action on those agents.

Efficiency

The local efficiency of a node indicates how well the network can transfer information when that node is removed. In some sense it describes how well the network functions when a node is eliminated. Attacker nodes with high efficiency are those nodes which will have little impact on the attacker's graph if they are removed, or if the cost for reaching them is increased. Consequently, we can choose either to attempt to decrease the overall efficiency of the attacker's graph by targeting defenses on nodes with high local efficiency, or we can target defenses on inefficient nodes to increase the cost of passing through bottlenecks in the attacker's graph. On the other hand, we want the defender's network to have high efficiency, so we might look for nodes with low local efficiency and then seek methods, such as adding additional nodes or edges, for improving it.

Application of Optimizations

Now, we consider applying these optimization approaches to our example problem to show how, in addition to modeling the impact of a defense, our dependency graph approach can also be used to identify areas for applying defenses. Since we have a limited set of defender options in this example, the impacts of which are summarized in Table 2.6, the multi-objective optimization described in section 2 is trivially solved as a full implementation of ASLR and DEP. The graph analysis approaches of section 2 are of more interest in this example. We consider edge betweenness centrality, local and global efficiency, and Girvan-Newman community detection on the graph in Figure 2.1. We begin with edge betweenness centrality. Recall from Section 2 that attacker edges with large centrality are attractive for targeting because they are the nodes most likely to appear along the attacker's shortest, and hence most likely, paths. The centrality for each attacker edge is presented in Table 2.7. From this, our analysis suggests defenses targeting edges a_7 and $a38$ or the edges leading to them. These edges lead to a collection of parallel paths beginning

with $a3, a8, a10$ and $a7$. Targeting these would be of little value, since the attacker would still be left with many alternative options. Now, since the recommended edges $a37$ and $a38$, which represent the attacker gaining knowledge of the machine instruction set and development of exploit code. These are not direclty impacted by ASLR or DEP, this suggests that a defense that does target those dependencies, such as instruction set randomization, would be useful in concert with ASLR and DEP [23, 2, 20]. Since the user's dependency graph has a single path the concept of betweenness centrality is not useful for analyzing it.

Table 2.7. Edge Betweenness Centrality for Attacker Edges

Edge	Centrality	Edge	Centrality
a0	0.08	**a7**	0.14
a1	0.15	**a8**	0.05
a2	0.20	**a9**	0.02
a3	0.05	**a10**	0.05
a4	0.02	**a11**	0.02
a5	0.15	**a37**	0.24
a6	0.08	**a38**	0.27

Next, we consider the graph efficiency metrics. Recall from Section 2 that nodes with high efficiency are those that will have little impact on the graph if they are removed. This implies that we should either target inefficient attacker nodes, helping to create bottlenecks in the attacker's graph, or to broadly target efficient nodes in an attempt to lower the global efficiency of the attacker's graph. Since the attacker's graph is consists almost entirely of serial connections, we find that node $n38$, which leads to set of parallel paths, has efficiency 0.25 and is the most efficient in the network and so preventing or increasing the attacker's difficulty for reaching it should be useful. Indeed, this is the only node in the graph with nonzero efficiency. As with edge betweenness centrality, this again suggests introducing instruction set randomization to make it more difficult for the attacker to learn the machine's instruction set and to use this knowledge to craft an exploit. This metric also reveals that the user's graph has low efficiency and so implementing measures for parallelizing paths in the user's graph might be useful. Additionally, since two nodes ($n5$ and $n6$) are shared by the users and attackers, any defenses that target the attacker's dependencies for reaching these nodes should be careful not to overly burden the user.

Now, recall from Section 2 that the Girvan-Newman algorithm, finds those edges that are most likely to be in between separate communities in a graph. By locating these edges, we can target defenses at the boundaries in between attacker communities to attempt to separate them from the graph or to make it more costly for the attacker to reach one community from the other. From the user's perspective, defenses for blurring the boundaries between communities may be desirable since they may make it easier to fulfill dependencies. Applying Girvan-Newman to the attacker's graph to produce four communities reveals (n1, n8), (n9, n5, n37), (n38, n10, n11, n12), and (n15, n6, n17) as the four attacker communities, with (n1, n8) the least important of these. Of the remaining communities, the boundary nodes are n9, n37, n38, and n15. This suggests that defenses that make it more difficult for an attacker to identify the machine architecture and instruction set, to develop exploit code, and to use buffer overflows to inject code are all of potential interest. ASLR impacts the use of buffer overflows by making it more difficult for an attacker to redirect program

execution to the location of the injected code. DEP also has an impact on buffer overflows since it makes it more likely that injected code will be in a non-executable region. Additionally, ISR is likely to increase attacker difficulty for learning about the instruction set and developing exploit code. Finally, we note that the three prominent communities detected by Girvan-Newman can be described as ”exploit development” (n9, n5, n37), ”exploit insertion” (n38, n10, n11, n12), and ”exploit execution” (n15, n6, n17), and so techniques for separating these communities, such as execution environment between exploit development or insertion and execution, may be useful.

Defense Discovery

The graph analysis algorithms suggest approaches where one might look to apply defenses to have the greatest impact on attackers. This permits ”defense discovery” in which we use analysis of our dependency graphs to find attacker and defender nodes and edges that are most suitable for targeting with a defense. In this setting, there may be situations in which an existing defense addresses the identified edges or nodes, such as with using ISR to frustrate exploit development in the preceding example, and other situations in which there is no existing defense that targets the identified nodes or edges. In these cases, the analysis provides some research direction into the identification of new defensive techniques that are likely to have a large impact on attackers without overly burdening legitimate users. These analysis could be applied to any dependency graph to help with identification of appropriate defenses or to suggest research directions for development of new defenses.

Chapter 3

State Space

Approach

We begin the modeling process by making several assumptions. We assume that the attacker and the defender can each influence the system state by applying inputs to the system. These inputs may be corrupted by noise. We further assume that attackers and defenders can at least partially observe the system state. These output observations may also be corrupted by noise. Additionally, we assume that there is no guarantee that either party can impact all of the state variables, or that their observations of the state are correct.

Open Loop

Figure 3.1 shows the simplest open-loop state space representation that conforms to these assumptions. It is a mixed deterministic-stochastic system approach for modeling the dynamic behavior of MTD systems. By inspecting Figure 3.1 we see that

$$\mathbf{x_{k+1}} = A\mathbf{x_k} + B_d\mathbf{u_k} + B_a\mathbf{m_k} + \mathbf{w_k} \tag{3.1}$$

$$\mathbf{y_k} = C_d\mathbf{x_k} + \mathbf{v_k} \tag{3.2}$$

$$\mathbf{z_k} = C_a\mathbf{x_k} + \mathbf{q_k} \tag{3.3}$$

where $\mathbf{x_k} \in \mathbb{R}^n$ is the state vector, $\mathbf{y_k} \in \mathbb{R}^{q_d}$ and $\mathbf{p_k} \in \mathbb{R}^{q_a}$ are the output vectors as observed by the defender and attacker, respectively, $\mathbf{u_k} \in \mathbb{R}^{p_d}$ and $\mathbf{m_k} \in \mathbb{R}^{p_a}$ are the defender and attacker's respective input vectors, $\mathbf{w_k}$ is a stochastic disturbance input, $\mathbf{v_k}$ and $\mathbf{q_k}$ are random vectors impacting the defender's and attacker's respective observations of the state, $A \in \mathbb{R}^{n\times n}$ is the system matrix, $B_d \in \mathbb{R}^{n\times p_d}$ and $B_a \in \mathbb{R}^{n\times p_a}$ are the input matrices for the defender and attacker, respectively, and $C_d \in \mathbb{R}^{q_d\times n}$ and $C_a \in \mathbb{R}^{q_a\times n}$ are the defender's and attacker's respective output matrices.

To represent an MTD with this model we must determine how to construct each of these matrices and vectors. We have previously developed a dependency graph based approach for analyzing cyber defenses [18]. In that approach different classes of actors, such as system users, administrators, and attackers, are each assumed to have some set of dependencies that they must fulfill in order to accomplish their goal. Some of these may be shared between the different classes of actors, while others will not be. Each of the dependencies has some costs associated with fulfilling

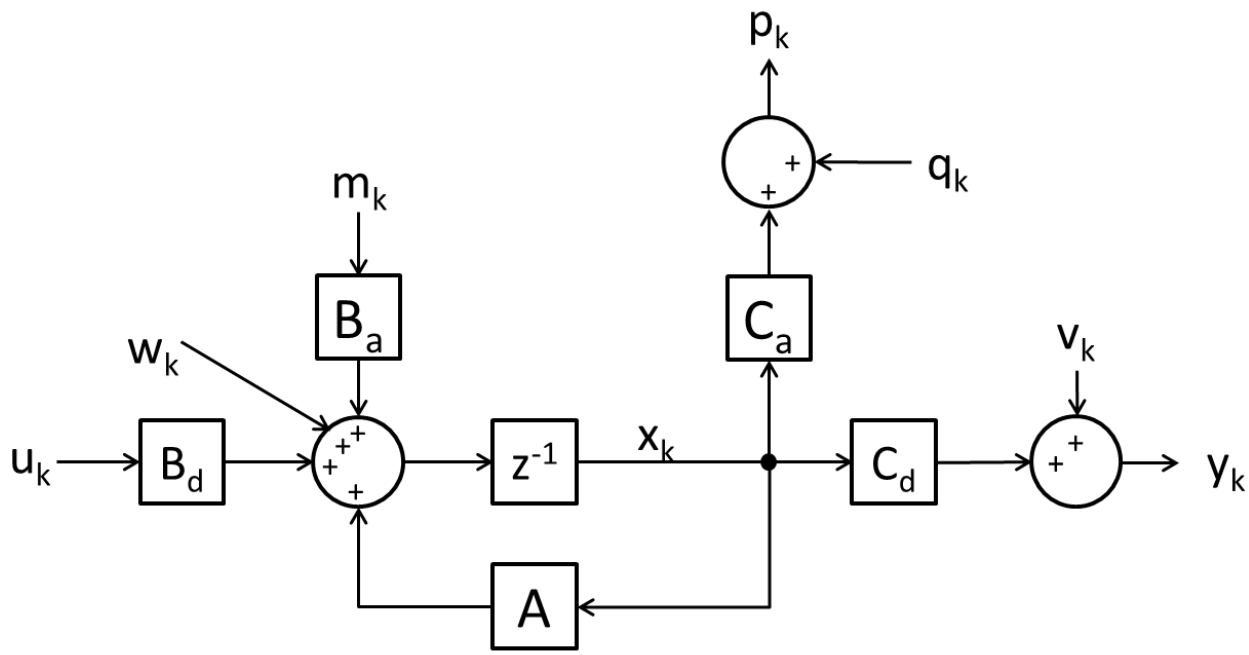

Figure 3.1. Open-loop state space representation for moving target defense

it. We interpret the state of the system at time k as a vector of costs for satisfying each of the n edges in the dependency graph. It is then natural that the system matrix, A, be an adjacency matrix representation of the dependency graph. Consequently, $a_{ij} = 1$ if there is an edge from node i to node j in the dependency graph, and $A_{ij} = 0$ otherwise. The $n \times p_d$ matrix B_d maps the defender's control input $\mathbf{u_k}$. One option is for the input $\mathbf{u_k}$ to define the change in each cost metric for each edge in the dependency graph as a result of some defender action. Consequently, we will have $p_d > n$. In this interpretation B_d will be a block diagonal matrix with its entries averaging the impact of the metrics to obtain an overall change in cost for an edge. For example, if there are six metrics and we use an unweighted average to combine them into a cost, then B_d will have the form

$$\begin{bmatrix} \alpha & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \alpha & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \alpha & \dots & \mathbf{0} & \dots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots & & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \alpha & \dots & \mathbf{0} \\ \vdots & \vdots & \vdots & & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} \end{bmatrix}$$

where

$$\alpha = \begin{bmatrix} 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \end{bmatrix}$$

$$\mathbf{0} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The attacker's input matrix B_a and input vector $\mathbf{m_k}$ are defined similarly, but will have nonzero entries along the entire diagonal. Alternatively, we can have $B_d \in \mathbb{R}^{n\times n}$ and $B_a \in \mathbb{R}^{n\times n}$, which will cause the input matrices to impact each metric equally. In this formulation, the input vectors $\mathbf{u_k}$ and $\mathbf{m_k}$ represent the change in cost for fulfilling a dependency. This form is more amenable to analysis of closed-loop feedback systems.

We interpret $\mathbf{y_k} = C_d\mathbf{x_k} + \mathbf{v_k}$ as the defender's view of the current system state $\mathbf{x_k}$. Consequently, each of the q_d rows of C_d will have a single 1 in a column corresponding to one of the dependency graph's edges. If $q_d < n$ then some of the edges are not visible to the defender. Oftentimes this will be the case, since we do not necessarily assume that the defender has knowledge of all of the attacker's dependencies. Similarly, the attacker's view of the system state is $\mathbf{z_k} = C_a\mathbf{x_k} + \mathbf{q_k}$ and so C_a also has a single 1 in each of its q_a rows. In general, we can have $q_a < n$ to indicate that the attacker has incomplete knowledge of the dependency graph. However, in accordance with Shannon's maxim that "the enemy knows the system" [31] we will usually assume that $q_a = n$ and that $C_a = I$, providing the attacker with visibility into each of the graph's edges. Note, however, that while the attacker may have knowledge of each of the edges, the random vector $\mathbf{q_k}$ may prevent the attacker from having perfect knowledge of $\mathbf{x_k}$.

We say that the system is *defender observable* if

$$\mathbb{O}_d = \begin{bmatrix} C_d & C_dA & C_dA^2 & \dots & C_dA^{n-1} \end{bmatrix}^T \tag{3.4}$$

has rank n and that it is *defender controllable* if

$$\mathbb{C}_d = \begin{bmatrix} B_d & AB_d & A^2B_d & \dots & A^{n-1}B_d \end{bmatrix} \tag{3.5}$$

has rank n. Similarly, the system is *attacker observable* if

$$\mathbb{O}_a = \begin{bmatrix} C_a & CaA & C_aA^2 & \dots & C_aA^{n-1} \end{bmatrix}^T \tag{3.6}$$

has rank n and it is *attacker controllable* if

$$\mathbb{C}_a = \begin{bmatrix} B_a & AB_a & A^2B_a & \dots & A^{n-1}B_a \end{bmatrix} \tag{3.7}$$

has rank n.

Now, consider the defender's view of the system's transfer function. Taking Z-transforms, we obtain

$$zX(z) - zx_0 = AX(z) + B_dU(z) + B_aM(z) + W(z)$$
$$Y(z) = C_dX(z) + V(z)$$

by rearranging the state equation and substituting into the expression for $Y(z)$ we obtain

$$Y(z) = C_d(zI - A)^{-1}zx_0 + C_d(zI - A)^{-1}\left[B_dU(z) + B_aM(z) + W(z)\right] + V(z) \tag{3.8}$$

The transfer function observed from the defender's perspective is $H_d(z) = Y(z)/U(z)$ and so

$$H_d(z) = C_d(zI - A)^{-1}\left[B_d + \frac{B_a M(z) + W(z)}{U(z)}\right] + V(z)/U(z) \tag{3.9}$$

Ideally for the defender the $B_a M(z)$ term will be null, indicating no attacker activity. The defender can estimate the value of this term from output measurements by solving eqn. 3.8 for $B_a M(z)$, which yields

$$B_a M(z) = C_d^{-1}(zI - A)\left[Y(z) - V(z)\right] - B_d U(z) - W(z) \tag{3.10}$$

Similarly, the transfer function from the attacker's perspective is $H_a(z) = P(z)/M(z)$. Manipulating the equations produces the attacker's view of the output as

$$P(z) = C_a(zI - A)^{-1}\left[B_d U(z) + B_a M(z) + W(z)\right] + Q(z) \tag{3.11}$$

and

$$H_a(z) = C_a(zI - A)^{-1}\left[B_a + \frac{B_d U(z) + W(z)}{M(z)}\right] + Q(z)/M(z) \tag{3.12}$$

The attacker can estimate the defender's influence on the system by solving eqn. 3.11 for $B_d U(z)$ to obtain

$$B_d U(z) = C_a^{-1}(zI - A)\left[P(z) - Q(z)\right] - B_a M(z) - W(z) \tag{3.13}$$

From both the attacker's and the defender's perspective, this open loop system is stable if the eigenvalues of A fall within the unit circle.

Output Feedback

Now, we add output feedback and set points to our representation, as shown in Figure 3.2. This allows the defender and the attacker to attempt to drive the system state, as they observe it, to some desired value. State feedback is not an option for defenders since in general the state is not observable to the defender, and in practice may not be visible to the attacker, either. Considering Figure 3.2 we first note that

$$\mathbf{x_{k+1}} = A\mathbf{x_k} + B_d\mathbf{u_k} + B_a\mathbf{m_k} + \mathbf{w_k} \tag{3.14}$$

$$\mathbf{y_k} = C_d\mathbf{x_k} + \mathbf{v_k} \tag{3.15}$$

$$\mathbf{z_k} = C_a\mathbf{x_k} + \mathbf{q_k} \tag{3.16}$$

where

$$\mathbf{u_k} = \mathbf{r_k} - K_d\mathbf{y_k} \tag{3.17}$$

$$\mathbf{m_k} = \mathbf{s_k} - K_a\mathbf{p_k} \tag{3.18}$$

and where we have

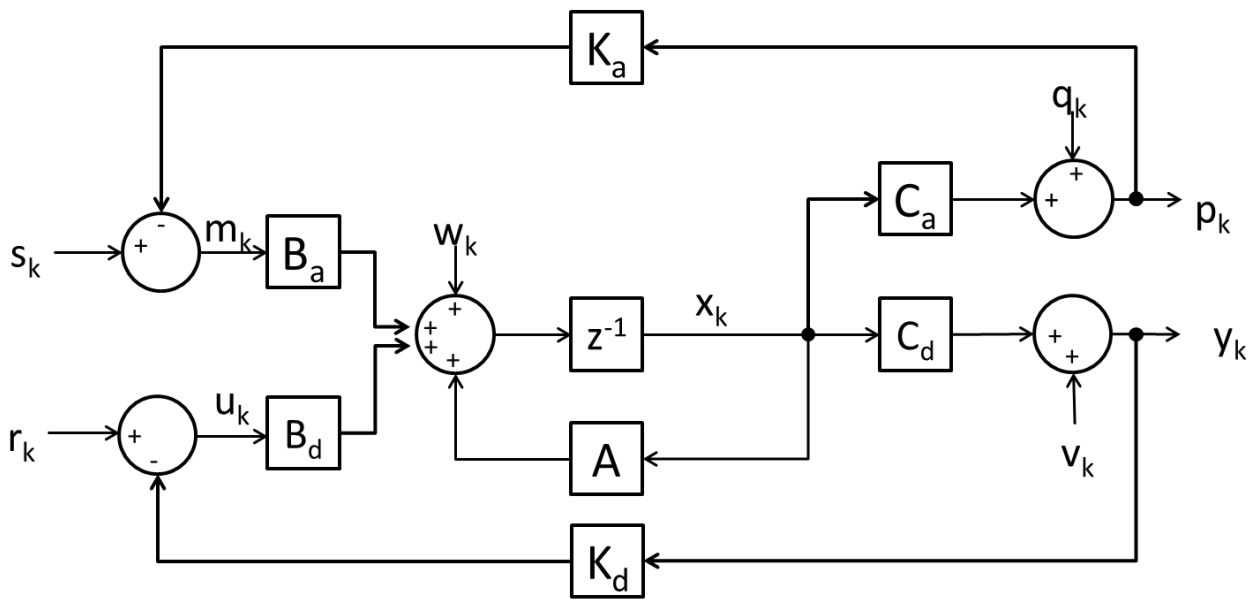

Figure 3.2. Output feedback state space representation for moving target defense.

$K_d \in \mathbb{R}^{p_d \times q_d}$ is the defender's feedback gain

$K_a \in \mathbb{R}^{p_a \times q_a}$ is the attacker's feedback gain

and where $\mathbf{r_k}$ and $\mathbf{s_k}$ are the defender and attacker set points, respectively. We begin by finding the defender and attacker transfer functions. Considering the defender first, we assume zero initial conditions and take z-transforms of the state and output

$$zX(z) = AX(z) + B_d U(z) + B_a M(z) + W(z)$$

$$U(z) = R(z) - K_d \left[C_d X(z) + V(z)\right]$$

$$Y(z) = C_d X(z) + V(z)$$

substituting $U(z)$ into the first expression and expanding and rearranging terms yields

$$X(z) = (zI - A + B_d K_d C_d)^{-1} \times \left[B_d R(z) + B_a M(z) - B_d K_d V(z) + W(z)\right] \quad (3.19)$$

Substituting this into the expression for $Y(z)$ produces

$$Y(z) = C_d (zI - A + B_d K_d C_d)^{-1} \times \left[B_d R(z) + B_a M(z) - B_d K_d V(z) + W(z)\right] + V(z) \quad (3.20)$$

and so we obtain the closed loop transfer function seen by the defender as

$$H_{cld}(z) = \frac{Y(z)}{R(z)} = \tag{3.21}$$

$$C_d\left(zI - A + B_d K_d C_d\right)^{-1}\left[B_d + \frac{B_a M(z) - B_d K_d V(z) + W(z)}{R(z)}\right] + \frac{V(z)}{R(z)}$$

Now, in the defender's ideal case the $B_a M(z)$ term is the null vector. The defender can use a measurement of $Y(z)$ to estimate the value of $B_a M(Z)$. Solving for $B_a M(Z)$ in eqn. 3.20 gives

$$B_a M(z) = C_d^{-1}\left(zI - A + B_d K_d C_d\right)\left[Y(z) - V(z)\right] - B_d R(z) + B_d K_d V(z) - W(z) \tag{3.22}$$

The defender may be able to use this estimate of the attacker's influence on the system to counteract the attacker's actions.

We may also desire to express the transfer function from R to Y while incorporating the attacker's input vector $\mathbf{m_k}$. If we assume knowledge of the attacker's behavior and design a control system that incorporates it, we may then be able to study how closely we can approximate this situation when the defender either does not have knowledge of the attacker's system, or when the defender can only estimate the attacker's influence, for instance, with eqn. 3.22. We begin by considering the system state and both defender and attacker inputs

$$zX(z) = AX(z) + B_d U(z) + B_a M(z) + W(z)$$

$$U(z) = R(z) - K_d\left[C_d X(z) + V(z)\right]$$

$$M(z) = S(z) - K_a\left[C_a X(z) + Q(z)\right]$$

Substituting the expressions for $U(z)$ and $M(Z)$ into the expression for $X(Z)$ we obtain

$$X(z) = \left(zI - A + B_d K_d C_d + B_a K_a C_a\right)^{-1} \times \left[B_d R(z) + B_a S(z) - B_d K_d V(z) - B_a K_a Q(z) + W(z)\right] \tag{3.23}$$

Now, since we have that $Y(z) = C_d X(z) + V(z)$ then the transfer function from R to Y is

$$\frac{Y(z)}{R(z)} = C_d\left(zI - A + B_d K_d C_d + B_a K_a C_a\right)^{-1} \times \left[B_d + \frac{B_a S(z) - B_d K_d V(z) - B_a K_a Q(z) + W(z)}{R(z)}\right] + \frac{V(z)}{R(z)} \tag{3.24}$$

which is the same as eqn. 3.21 when B_a is the zero matrix. Finally, we also note from eqn. 3.23 that the system is stable if the roots of $zI - A + B_d K_d C_d + B_a K_a C_a$ are within the unit circle.

We can determine the attacker's transfer function $Y_{cla} = \frac{P(z)}{S(z)}$ by taking z-transforms of the state and output, substituting the expression for $M(z)$ into the expression for $X(z)$, expand and rearrange terms and substitute the result into the expression for $P(z)$ to obtain

$$P(z) = C_a\left(zI - A + B_aK_aC_a\right)^{-1} \times \\ \left[B_dU(z) + B_aS_z - B_aK_aQ(z) + W(z)\right] + Q(z) \quad (3.25)$$

and so the attacker's closed loop transfer function is

$$H_{cla}(z) = \frac{P(z)}{S(z)} = \quad (3.26)$$

$$C_a\left(zI - A + B_aK_aC_a\right)^{-1}\left[B_a + \frac{B_dU(z) - B_aK_aQ(z) + W(z)}{S(z)}\right] + \frac{Q(z)}{S(z)}$$

The attacker can use eqn. 3.25 to estimate the defender's influence on the system, $B_dU(z)$, by calculating

$$B_dU(z) = C_a^{-1}\left(zI - A + B_aK_aC_a\right)\left[P(z) - Q(z)\right] - B_aS(z) + B_aK_aQ(z) - W(z)$$

Now, let's consider the controllability and observability of this system. We begin by writing

$$\mathbf{u_k} = \mathbf{r_k} - K_d\mathbf{y_k} = \mathbf{r_k} - K_d\left(C_d\mathbf{x_k} + \mathbf{v_k}\right) \quad (3.27)$$

$$\mathbf{m_k} = \mathbf{s_k} - K_a\mathbf{p_k} = \mathbf{s_k} - K_a\left(C_a\mathbf{x_k} + \mathbf{q_k}\right) \quad (3.28)$$

considering the defender first, we substitute this expression for $\mathbf{u_k}$ into the expression for $\mathbf{x_{k+1}}$ to obtain

$$\mathbf{x_{k+1}} = A\mathbf{x_k} + \left(B_d\mathbf{r_k} - B_dK_dC_d\mathbf{x_k} - B_dK_d\mathbf{v_k}\right) + B_a\mathbf{m_k} + \mathbf{w_k}$$

grouping terms provides

$$\left(A - B_dK_dC_d\right)\mathbf{x_k} + B_d\left(\mathbf{r_k} - K_d\mathbf{v_k}\right) + B_a\mathbf{m_k} + \mathbf{w_k}$$

and so the defender's observability and controllability matrices are

$$\mathbb{O}_d = \left[\begin{array}{cccc} C_d & C_d\left(A - B_dK_dC_d\right) & \ldots & C_d\left(A - B_dK_dC_d\right)^{n-1} \end{array}\right]^T \quad (3.29)$$

$$\mathbb{C}_d = \left[\begin{array}{cccc} B_d & \left(A - B_dK_dC_d\right)B_d & \ldots & \left(A - B_dK_dC_d\right)^{n-1}B_d \end{array}\right] \quad (3.30)$$

Considering the attacker's perspective, we substitute the expression for $\mathbf{m_k}$ into the expression for $\mathbf{x_{k+1}}$ from which the attacker's observability and controllability matrices are

$$\mathbb{O}_a = \left[\begin{array}{cccc} C_a & Ca\left(A - B_aK_aC_a\right) & \ldots & C_a\left(A - B_aK_aC_a\right)^{n-1} \end{array}\right]^T \quad (3.31)$$

$$\mathbb{C}_a = \left[\begin{array}{cccc} B_a & \left(A - B_aK_aC_a\right)B_a & \ldots & \left(A - B_aK_aC_a\right)^{n-1}B_a \end{array}\right] \quad (3.32)$$

Here, the system is defender stable if the eigenvalues of $A - B_dK_dC_d$ are within the unit circle. The system is attacker stable if the eigenvalues of $A - B_aK_aC_a$ are within the unit circle.

State Feedback

To overcome the defender's inability to measure the current state, we can also consider state observer feedback representations, as shown in 3.3. Before beginning, we note that the defender may not be aware of all of the attacker's edges, and that this prevents the defender from having an observable system. Due to this, if the defender constructs a state observer feedback system then information about unobservable modes will not be included in the output prediction error term, and the defender's state observer will only be stable if all of the unobservable modes are stable [27].

Considering Fig. 3.3, we first write down the state equations as

$$\mathbf{x_{k+1}} = A\mathbf{x_k} - D_d K_d \widehat{\mathbf{x}}_{\mathbf{d,k}} + B_d \mathbf{r_k} - B_a K_a \widehat{\mathbf{x}}_{\mathbf{a,k}} + B_a \mathbf{s_k} + \mathbf{w_k} \tag{3.33}$$

$$\widehat{\mathbf{x}}_{\mathbf{d,k+1}} = A\widehat{\mathbf{x}}_{\mathbf{d,k}} + B_d \mathbf{u_k} + L_d \left(\mathbf{y_k} - C_d \widehat{\mathbf{x}}_{\mathbf{d}}\right) \tag{3.34}$$

$$\widehat{\mathbf{x}}_{\mathbf{a,k+1}} = A\widehat{\mathbf{x}}_{\mathbf{a,k}} + B_a \mathbf{m_k} + L_a \left(\mathbf{p_k} - C_a \widehat{\mathbf{x}}_{\mathbf{a}}\right) \tag{3.35}$$

$$\mathbf{y_k} = C_d \mathbf{x_k} + \mathbf{v_k} \tag{3.36}$$

$$\mathbf{p_k} = C_a \mathbf{x_k} + \mathbf{q_k} \tag{3.37}$$

$$\mathbf{u_k} = -K_d \widehat{\mathbf{x}}_{\mathbf{d}} + \mathbf{r_k} \tag{3.38}$$

$$\mathbf{m_k} = -K_a \widehat{\mathbf{x}}_{\mathbf{a}} + \mathbf{s_k} \tag{3.39}$$

where $\widehat{\mathbf{x}}_{\mathbf{d,k}}$ and $\widehat{\mathbf{x}}_{\mathbf{a,k}}$ are the defender's and the attacker's estimates of the state at time k, respectively and

$L_d \in \mathbb{R}^{n \times q_d}$ is the defender's observer gain

$L_a \in \mathbb{R}^{n \times q_a}$ is the attacker's observer gain

We begin by finding the transfer function from the defender's perspective. First, we take the z-transform of eqn. 3.33 and solve for $X(z)$, to obtain

$$X(z) = (zI - A + B_d K_d)^{-1} \times \left[B_d K_d E_d(z) + B_d R(z) + W(z) + B_a M(z)\right] \tag{3.40}$$

Substituting this into the z-transform of eqn. 3.36 and divide by $R(z)$ to find

$$\frac{Y(z)}{R(z)} = C_d \left(zI - A + B_d K_d\right)^{-1} \times \left[B_d + \frac{B_d K_d E_d(z) + W(z) + B_a M(z)}{R(z)}\right] + \frac{V(z)}{R(z)} \tag{3.41}$$

From the defender's perspective, this system is stable if the roots of $zI - A + B_d K_d$ are within the unit circle. The defender's observability and controllability matrices are

$$\mathbb{O}_d = \left[\begin{array}{cccc} C_d & C_d\left(A - B_d K_d\right) & \dots & C_d\left(A - B_d K_d\right)^{n-1} \end{array}\right]^T \tag{3.42}$$

$$\mathbb{C}_d = \left[\begin{array}{cccc} B_d & \left(A - B_d K_d\right) B_d & \dots & \left(A - B_d K_d\right)^{n-1} B_d \end{array}\right] \tag{3.43}$$

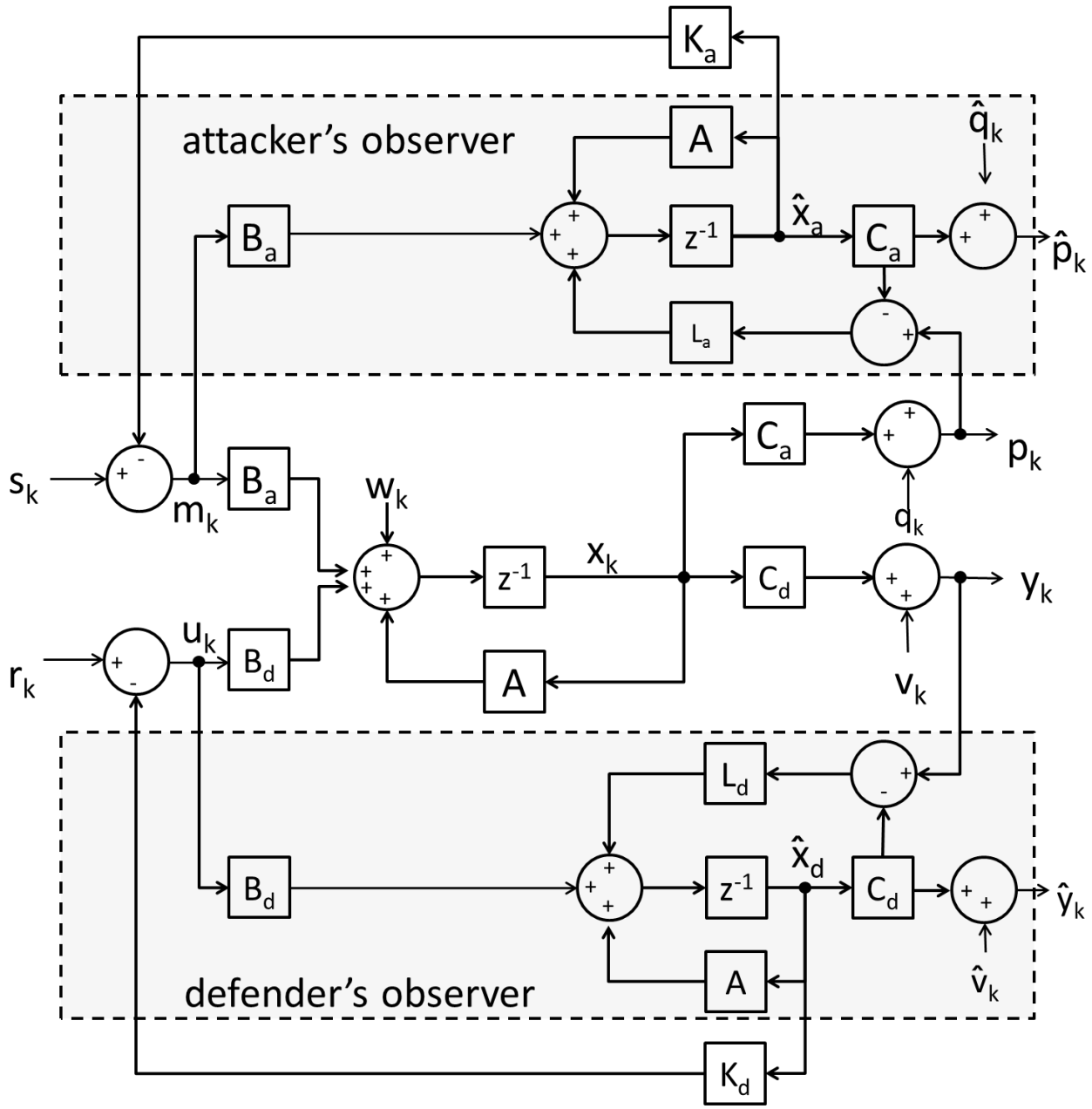

Figure 3.3. State feedback with state observer state space representation for moving target defense

By substituting eqn. 3.40 into eqn 3.36 and solving for $B_aM(z)$ we can obtain the defender's estimate of the attacker's influence on the system as

$$B_aM(z) = C_d^{-1}\left(zI - A + B_dK_d\right) \times \left[Y(z) - V(z)\right] - B_dK_dE_d(z) - B_dR(z) - W(z) \quad (3.44)$$

Similarly, we can obtain the transfer function from the attacker's perspective as

$$\frac{P(z)}{S(z)} = C_a\left(zI - A + B_aK_a\right)^{-1} \times \left[B_a + \frac{B_aK_aE_a(z) + W(z) + B_dU(z)}{S(z)}\right] + \frac{Q(z)}{S(z)} \quad (3.45)$$

From the attacker's perspective the system is stable if the roots of $zI - A + B_aK_a$ are within the unit circle, and the attacker's observability and controllability matrices are

$$\mathbb{O}_a = \left[\begin{array}{cccc} C_a & C_d\left(A - B_aK_a\right) & \ldots & C_a\left(A - B_aK_a\right)^{n-1} \end{array}\right]^T \quad (3.46)$$

$$\mathbb{C}_a = \left[\begin{array}{cccc} B_a & \left(A - B_aK_a\right)B_a & \ldots & \left(A - B_aK_a\right)^{n-1}B_a \end{array}\right] \quad (3.47)$$

The attacker can use measurements of $P(z)$ to estimate the defender's influence by

$$B_dU(z) = C_a^{-1}\left(zI - A + B_aK_a\right) \times \left[P(z) - Q(z)\right] - B_aK_aE_a(z) - B_aS(z) - W(z) \quad (3.48)$$

Now, let's consider the situation in which the defender has knowledge of the attacker's use of state-observer feedback. If we consider the error terms

$$\mathbf{e_{d,k}} = \mathbf{x_k} - \widehat{\mathbf{x}}_{\mathbf{d,k}}$$

$$\mathbf{e_{a,k}} = \mathbf{x_k} - \widehat{\mathbf{x}}_{\mathbf{a,k}}$$

then we can write the state estimates as

$$\widehat{\mathbf{x}}_{\mathbf{d,k}} = \mathbf{x_k} - \mathbf{e_{d,k}} \quad (3.49)$$

$$\widehat{\mathbf{x}}_{\mathbf{a,k}} = \mathbf{x_k} - \mathbf{e_{a,k}} \quad (3.50)$$

Substituting eqns. 3.49 and 3.50 into eqn. 3.33 produces

$$\mathbf{x_{k+1}} = \left(A - B_dK_d\right)\mathbf{x_k} + B_dK_d\mathbf{e_{d,k}} - B_aK_a\mathbf{x_k} + B_aK_a\mathbf{e_{a,k}} + B_d\mathbf{r_k} + B_a\mathbf{s_k} + \mathbf{w_k} \quad (3.51)$$

Now, by taking the z-transform of eqn. 3.51 and substituting into the z-transform of eqn. 3.36 we obtain the defender's transfer function

$$\frac{Y(z)}{R(z)} = C_d\left(zI - A + B_dK_d + B_aK_a\right)^{-1} \times \left[B_d + \frac{B_aS(z) + B_dK_dE_d(z) + B_aK_aE_a(z) + W(z)}{R(z)}\right] + \frac{V(z)}{R(z)} \quad (3.52)$$

The defender can obtain an estimate of the attacker's influence by substituting z-transforms of eqns. 3.39 and 3.50 into eqn. 3.44, solving for the B_a terms, and substituting $X(z) = C_d^{-1}(Y(z) - V(z))$ which yields

$$B_a\left[S(z) + K_a E_a(z) - K_a C_d^{-1}(Y(z) - V(z))\right] = C_d^{-1}(zI - A + B_d K_d) \times [Y(z) - V(z)] - B_d K_d E_d(z) - B_d R(z) - W(z) \quad (3.53)$$

Output Feedback Example: Application to Network Randomization Approaches

Now, we apply the output feedback modeling approach to studying network randomization techniques. In particular, we consider IP and port hopping and route randomization. We use the output feedback model to study the dynamics of a system, as viewed by defenders and attackers of that system, in response to defender and attacker actions on the system. This study allows us to explore the effects that attackers and defenders can have on the system and the dynamics of various defender strategies for deploying the mitigations.

First, we must combine the dependency graphs of the three defenses into a single dependency graph. Then, we take the adjacency matrix representation of this graph as the system matrix, A. As in section 3, the B_d and B_a matrices map the defender's and attacker's inputs, which represent the impact of the inputs on each of the metrics affiliated with each of the edges in the dependency graph, to their effect on the system state, x_k, which represents the cost of fulfilling the dependency represented by an edge in the graph. These input matrices define how the attacker and defender actions can influence the system state. Consequently, it is through the B_d matrix that we define the impact of a moving target defense on the system, In the output feedback system, we additionally have attacker and defender feedback gain matrices K_d and K_a. These matrices determine how much influence the defender's and attacker's measurement of the system state have on the selection of the next defender or attacker action.

The defender's input set point, r_k and attacker's input set point, s_k, represent the desired values of each of the metrics for each of the edges from the defender and attacker perspectives. While more nuanced set points are possible, we generally assume that the defender will want to minimize all of the costs for fulfilling all of the defender's dependencies while also maximizing all of the costs for fulfilling each of the attacker's dependencies. The attacker will desire to minimize each of the attacker's costs. While it is unlikely that either the defender or the attacker will be able to reach these ideal set points, the defender can use the distance from this set point, as measured by u_k, to help in selection of an appropriate moving target defense. That is, we envision a playbook, consisting of a set of j matrices $\left\{B_{d0} \ldots B_{dj}\right\}$ where j is the number of defensive moves available to the defender. Each of these defenses will impact the system by changing some of the cost metrics for some of the edges in the dependency graph. At each time step k the defender can choose which defense, if any, to apply to the system. If no defensive move is made then the defender

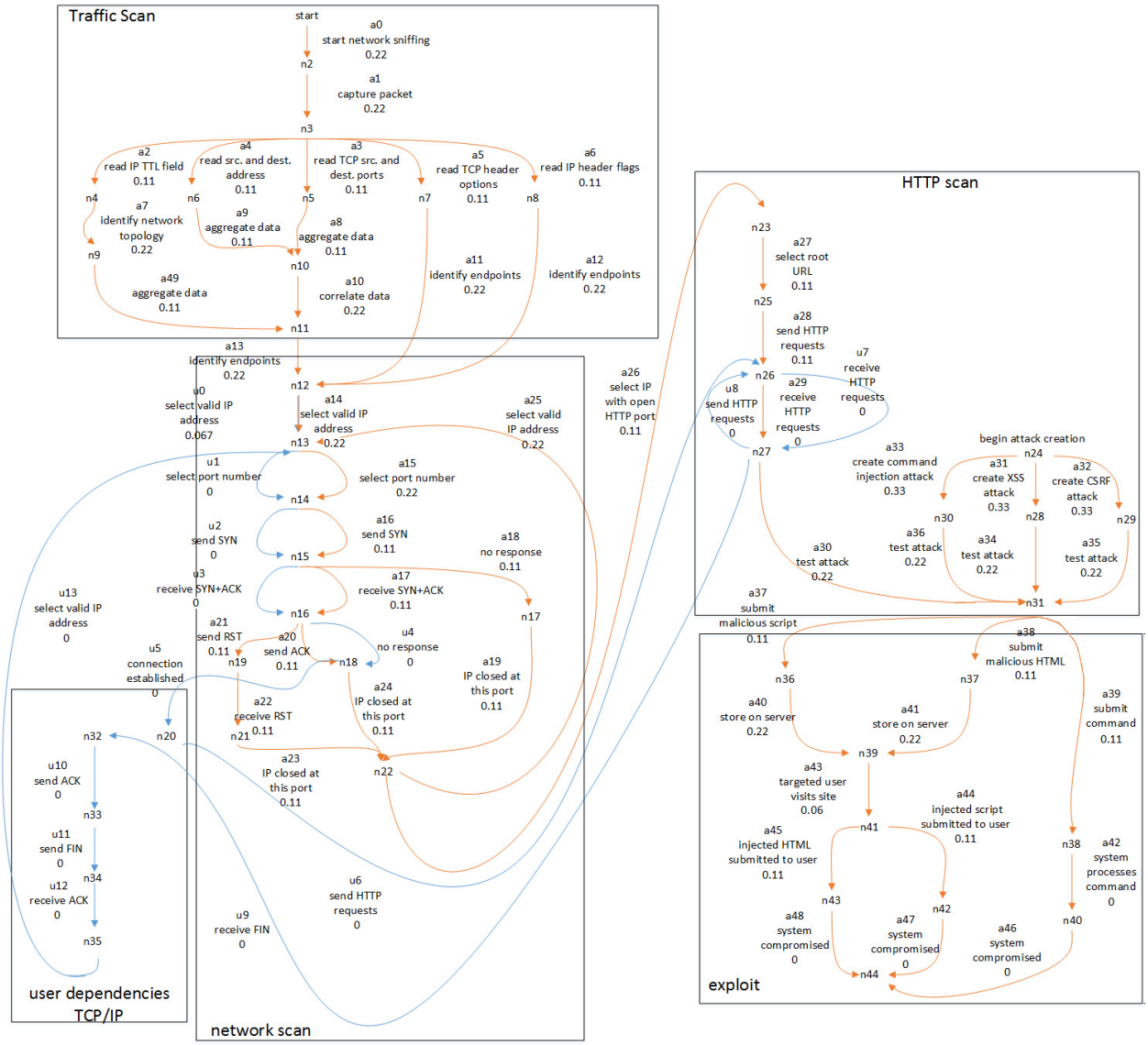

Figure 3.4. Network randomization dependency graph showing costs to fulfill dependencies prior to applying network randomization. The user's costs are all 0 because our user costs metrics consider percent increase in costs.

simply selects the initial input matrix B_{d0}, which represents the cost of fulfilling dependencies with no defenses in place, and has no impact on the system at that time step. Considering this, we have time-varying input matrices $B_d(k)$ and $B_a(k)$, and our problem is to identify which of the available defender moves $\{B_{d0} \dots B_{dj}\}$ to select at each time step. We call a sequence of moves $(B_d(0) \dots B_d(k))$ a *defender strategy*. There are many ways to undertake the analysis. We can define a strategy and then study how it performs by solving the state equations to find the resulting system dynamics. We can also use the defender's knowledge of the system at a given time k, as provided by the output vector y_k, to choose the next defense $B_{d,k+1}$ according to some rulebook and study the resulting evolution of the system dynamics. The resulting sequence of input matrices is a defender strategy guided by the rulebook, and so its performance can be used to help us define acceptable rulebooks and strategies for various attacker scenarios. Similar comments and definitions hold for the attacker.

Figure 3.4 shows an example dependency graph for standard TCP/IP and HTTP network communications on which we will study network randomization approaches [8]. The costs for fulfilling dependencies in this graph are the initial costs without any network randomizations in place. We begin by briefly describing the dependencies. First, the attacker performs a traffic scan to identify endpoints and valid IP addresses, a network scan to identify IP addresses with open HTTP ports, and finally an HTTP scan. The attacker also creates and tests attacks, and then launches an exploit by submitting a malicious script, HTML, or command to the server and injecting it to the targeted user when the user visits the site. This compromises the user's system. The user has dependencies related to communicating with HTTP over TCP/IP. The network randomization approaches impact the user and attacker dependencies in various ways. IP randomization impacts the attacker's ability to identify the network topology and correlate traffic scan data to identify valid endpoints. This makes it more difficult to select a valid IP address, which in turn increases the difficulty of finding an open HTTP port over which to submit a malicious payload to a targeted user. It also increases the user's cost for select valid IP addresses, communicating over TCP/IP, and sending HTTP requests. The impact on the user is primarily from increased network latency, and from small increases in CPU and memory requirements. Port randomization also complicates identification of the network topology, increases the difficulty of selecting valid HTTP ports, and consequently of submitting malicious payloads to the user. It increases the user's costs for selecting valid ports, communicating over TCP/IP, and sending HTTP requests by increasing network latency and slightly impacting CPU and memory requirements. Path randomization primarily impacts the attacker's ability to correlate traffic scan data for network mapping. It increases the user's costs for communicating over TCP/IP and sending HTTP requests by increasing network latency. When we combine these mitigations we impact the union of the dependencies impacted by the individual dependencies, although the change in costs for fulfilling the dependencies is usually less than the sum of the change for the individual mitigations.

We study this system with our output feedback state space model. For brevity, we consider only the influence of defensive actions through the B_d matrices and the defender's set point $\mathbf{r_k}$, although similar assignments hold for the attacker's input matrices B_a and set point $\mathbf{s_k}$.

We begin by transforming our dependency graph model of the system depicted in 3.5 into our output feedback state space representation. In this representation the system matrix $A \in \mathbb{R}^{n \times n}$

where $n = 44$ represents the 44 nodes in our system and the edges connecting these nodes. We also have input matrices $B_d, B_a \in \mathbb{R}^{44\times 44}$. The system's initial conditions represent the cost for fulfilling each of the dependencies without any defenses in use. For this example, we consider IP randomization. This defense impacts the defender by slightly increasing the memory and processing requirements for selecting valid IP addresses at $u0$ and $u13$, while also slightly degrading the system and network stability for doing so. It also increases latency for standard TCP/IP communications at $u2-u4$, $u6$, $u8$, and $u10-12$. From the attacker's perspective the time and cost for acquiring access and knowledge to identify the network topology ($a7$), correlate traffic scan data ($a10$), and select valid IP addresses ($a14$) is increased, as is the the unpredictability and frequency of defense movement at these edges. The unpredictability, frequency of movement, and time and cost for acquiring the necessary knowledge to select an IP address with open HTTP port($a26$), submit a malicious script, HTTP, or command ($a37-39$), and identify when the targeted user visits the compromised web site ($a43$) are also increased. We represent these changed costs for fulfilling dependencies by creating a new input matrix B_{d1} with diagonal entries reflecting the new overall costs for fulfilling each dependency. Note that since several edges can originate at the same node, as with node $n3$, the influence of a defense on a particular node is represented by adding the influence on all of the edges originating at that node. This is a consequence of our adjacency matrix representation of the graph, which causes the state space models to view the system from the perspective of costs for reaching a node, rather than the costs for traversing edges. This aliases the influence of edges originating from a single node into a single entry of the Bd or B_a matrices, and also into a single entry of the set point, state, and output vectors. The defender wants the cost of fulfilling every attacker edge to reach its maximum value of 1 and for the cost of fulfilling each defender edge to reach its minimum value of 0. These desires are codified in the set point vector $\mathbf{r_k}$ by setting the ith element of $\mathbf{r_k}$ to γ where γ is the number of attacker edges originating at node i. We also have output matrices $C_d, C_a \in \mathbb{R}^{n\times n}, n = 44$. We generally assume that the attacker has full knowledge of the dependency graph, and so $C_a = I_n$. In this case, we also assume that the defender is aware of all of the nodes in the dependency graph, and so we also have $C_d = I_n$. However, this will not always be the case. In particular, if we want to model an attack that the defender is not familiar with then we would include nodes and edges in the dependency graph that are unknown to the defender, and the diagonal elements of C_d that correspond to these nodes would be set to 0.

With these assignments in place we can begin to study the system. First, we consider properties of the open-loop system with B_{d0} in place, indicating that none of the network randomization options is applied. First, we find that the system is defender observable, but that it has 14 states that are uncontrollable by the defender. A pole-zero plot reveals that the open loop system is unstable, but we also find that all of the uncontrollable modes are located at the origin, and so we know that the system can be stabilized by the feedback controller. We use the Linear Quadratic Regulator (LQR) design technique to find a feedback gain matrix $K_d \in \mathbb{R}^{n\times n}, n = 44$ that stabilizes the closed-loop system [1]. The feedback has no impact on the observability or controllability of the system. We next find the time-domain response of the open and closed loop system and compare them. Results for the attacker edge $(n25, n26)$ are presented in figure 3.5. In figure 3.5a we see the impact of IP randomization on attacker dependency $A28 = (n28, n26)$, which is not directly influenced by the IP randomization. Here, we see that the IP randomization moves the the attacker's costs closer to the set point than the system with no network randomization. Similar behavior is observed in figure 3.5b, which shows the dynamics of attacker dependencies $A25$ and

$A26$. Here, edge $A26$ is directly influenced by the IP randomization but $A25$ is not. As before, the system that includes IP randomization is closer to obtaining the set point. Notice that, since two attacker edges originate at node $n22$ the set point in figure 3.5b is 2. While we do not show them here, many of the attacker and user dependencies do not exhibit any dynamic behavior under the baseline or IP randomization conditions. Some of these dependencies are uncontrollable and some are not influenced by these particular inputs. This will usually be the case. If the defender wishes to impact all of the attacker's dependencies, then the defender must identify a set of defenses that collectively cover all of the attacker's edges. Oftentimes, this will not be possible, and so we can instead use graph analysis techniques such as graph centrality and community detection to identify those attacker edges that are most beneficial to target. Further discussion on this topic appears in [18].

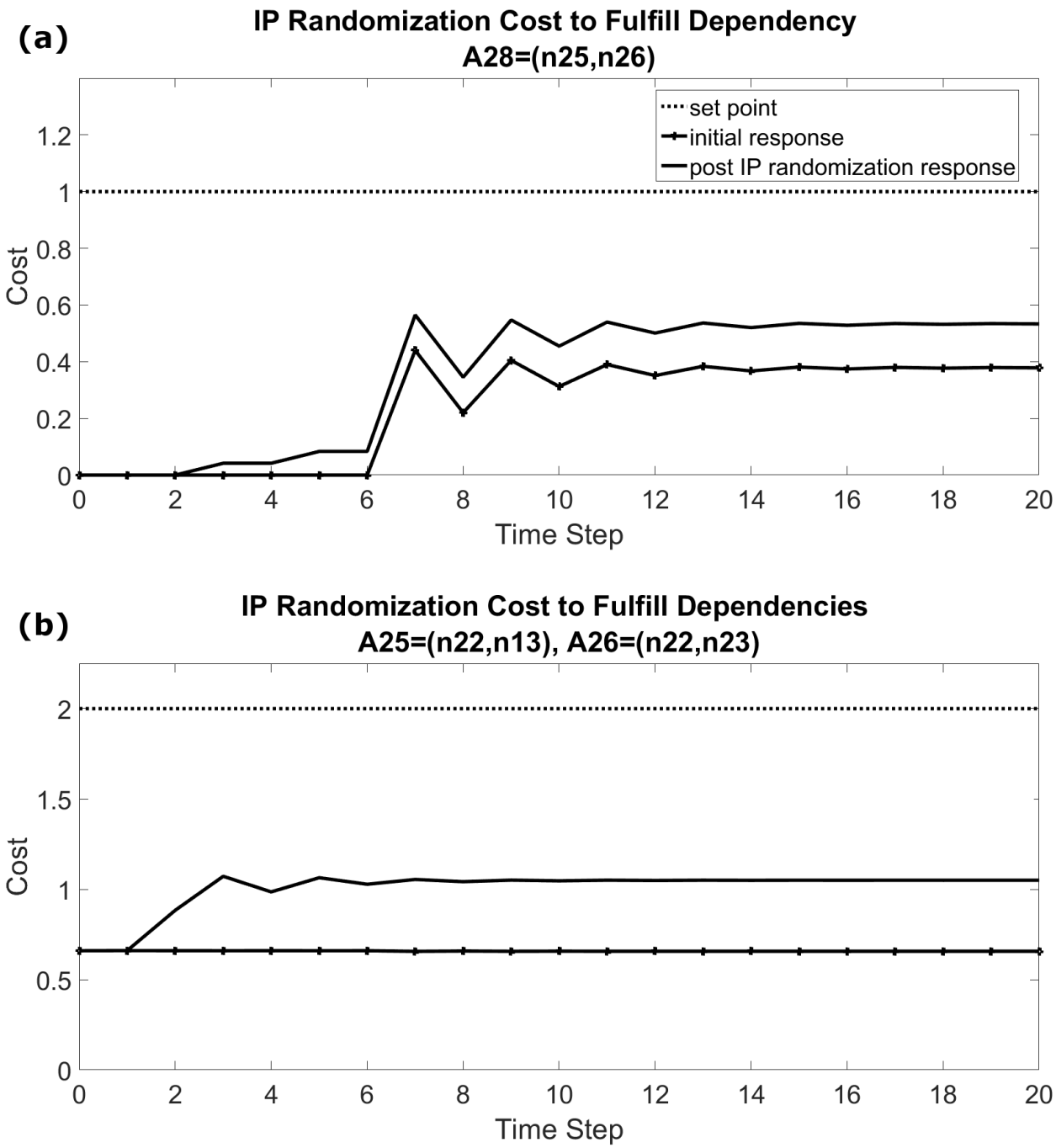

Figure 3.5. (a) Response for adversary dependencies $A28 = (n28, n26)$ and (b) $A25 = (n22, n13)$ and $A26 = (n22, n23)$ before and after IP randomization. Edges $A25$ and $A28$ are not directly impacted by the IP randomization.

Chapter 4

Conclusion

Although many MTD approaches have been presented in the literature, we find that relatively few of them have been adopted in practice. We suspect that this is because, in addition to impacting attackers, many MTDs break or increase the cost of system dependencies for users and administrators. These impacts on legitimate system users result in the cost of the MTD outweighing its benefits and prevents it from being adopted. To explore this issue we presented a dependency graph approach for modeling MTDs and their impacts on users, defenders, and attackers, and applied this model to address space layout randomization. We find that results from our model agree with previously reported experimental results, and so we then suggest optimization and graph analysis approaches for studying dependency graphs to identify appropriate locations for introducing MTDs. We have also introduced several state space representations for modeling the dynamic interactions between attackers and defenders in cyber systems. These modeling approaches may be particularly well suited to analyzing moving target defense systems since in these systems the defensive posture changes over time. We focused our attention on describing several potential models and developing the mathematical fundamentals for analyzing them, including various transfer functions, observability, and controllability. Future work can use this foundation to explore mode advanced concepts by adopting concepts from the wealth of existing control theory work to these new models.

Future work should also study the resiliency of systems to perturbations and modeling inaccuracies. In particular, it will be important to understand how sensitive the defender is to inaccuracies in modeling the attacker. For example, sensitivity to an attacker that uses a different feedback structure than that modeled by the defender, or an attacker that uses an open-loop strategy in which attacks are not influenced by observation of the system behavior are both of interest. It will also be important to study sensitivity to inaccuracies in the dependency graph. These may be due to unknown and un-modeled dependencies or inaccurately modeled dependencies. Additionally, our network randomization example only considers the defender's influence on the system. Future analysis should study the system dynamics that result from a combination of defender and attacker inputs to the system.

References

[1] Panos J Antsaklis and Anthony N Michel. *Linear systems*. Springer Science & Business Media, 2006.

[2] et al Barrantes, Elena Gabriela. Randomized instruction set emulation. In *ACM Transactions on Information and System Security (TISSEC) 8.1*, pages 3–40. ACM, 2005.

[3] Ulrik Brandes. On variants of shortest-path betweenness centrality and their generic computation. *SOCIAL NETWORKS*, 30(2), 2008.

[4] Roderic Broadhurst, Peter Grabosky, Mamoun Alazab, Brigitte Bouhours, and Steve Chon. An analysis of the nature of groups engaged in cyber crime. *An Analysis of the Nature of Groups engaged in Cyber Crime, International Journal of Cyber Criminology*, 8(1):1–20, 2014.

[5] L. Carin, G. Cybenko, and J. Hughes. Cybersecurity strategies: The queries methodology. *Computer*, 41(8):20–26, Aug 2008.

[6] Kevin M Carter, James F Riordan, and Hamed Okhravi. A game theoretic approach to strategy determination for dynamic platform defenses. In *Proceedings of the First ACM Workshop on Moving Target Defense*, pages 21–30. ACM, 2014.

[7] Marco M Carvalho, Thomas C Eskridge, Larry Bunch, Jeffrey M Bradshaw, Adam Dalton, Paul Feltovich, James Lott, and Daniel Kidwell. A human-agent teamwork command and control framework for moving target defense (mtc2). In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, page 38. ACM, 2013.

[8] Adrian Chavez, William Stout, and Sean Peisert. Techniques for the dynamic randomization of network attributes. In *Proceedings of the 49th Annual International Carnahan Conference on Security Technology*, 2015.

[9] Kim-Kwang Raymond Choo. The cyber threat landscape: Challenges and future research directions. *Computers & Security*, 30(8):719–731, 2011.

[10] Richard Colbaugh and Kristin Glass. Predictability-oriented defense against adaptive adversaries. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 2721–2727. IEEE, 2012.

[11] Sean Crosby, Marco Carvalho, and Daniel Kidwell. A layered approach to understanding network dependencies on moving target defense mechanisms. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, page 36. ACM, 2013.

[12] Michael Crouse, Bryan Prosser, and Errin W. Fulp. Probabilistic performance analysis of moving target and deception reconnaissance defenses. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 21–29, New York, NY, USA, 2015. ACM.

[13] George Cybenko and Jeff Hughes. No free lunch in cyber security. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 1–12, New York, NY, USA, 2014. ACM.

[14] Dino Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010.

[15] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[16] David Patrick Duggan. Generic threat profiles. Technical Report SAND2005-5411, Sandia National Laboratories, 2005.

[17] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[18] Jason Hamlet and Christopher Lamb. Dependency graph analysis and moving target defense selection. In *submitted*, 2016.

[19] Thomas Hobson, Hamed Okhravi, David Bigelow, Robert Rudd, and William Streilein. On the challenges of effective movement. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 41–50, New York, NY, USA, 2014. ACM.

[20] et al. Hu, Wei. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments.* ACM, 2006.

[21] Jafar Haadi H. Jafarian, Ehab Al-Shaer, and Qi Duan. Spatio-temporal address mutation for proactive cyber agility against sophisticated attackers. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 69–78, New York, NY, USA, 2014. ACM.

[22] Stephen Jones, Alexander Outkin, Jared Gearhart, Jacob Hobbs, John Siirola, Cindy Phillips, Stephen Verzi, Daniel Tauritz, Samuel Mulder, and Asmeret Naugle. Evaluating moving target defense with pladd. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2015.

[23] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security.* ACM, 2003.

[24] Erik Miehling, Mohammad Rasouli, and Demosthenis Teneketzis. Optimal defense policies for partially observable spreading processes on bayesian attack graphs. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 67–76, New York, NY, USA, 2015. ACM.

[25] Erik Miehling, Mohammad Rasouli, and Demosthenis Teneketzis. Optimal defense policies for partially observable spreading processes on bayesian attack graphs. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 67–76. ACM, 2015.

[26] Hamed Okhravi, Thomas Hobson, David Bigelow, and William Streilein. Finding focus in the blur of moving-target techniques. *Security & Privacy, IEEE*, 12(2):16–26, 2014.

[27] Alan V Oppenheim and George C Verghese. *Signals, systems and inference*. Prentice Hall, 2015.

[28] Michael Orlando. Defending microsoft windows against 0-day exploits using emet. In *ICS CERT Industrial Control Systems Joint Working Group Meeting, Fall*, 2012.

[29] Achintya Prakash and Michael P Wellman. Empirical game-theoretic analysis for moving target defense. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 57–65. ACM, 2015.

[30] Chris Rohlf and Yan Ivnitskiy. Attacking clientside jit compilers. *Black Hat USA*, 2011.

[31] Claude E Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.

[32] Alexander Sotirov. Bypassing memory protections: The future of exploitation. In *USENIX Security*, 2009.

[33] William Stallings. *Handbook of computer-communications standards; Vol. 1: the open systems interconnection (OSI) model and OSI-related standards*. Macmillan Publishing Co., Inc., 1987.

[34] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[35] Marten Van Dijk, Ari Juels, Alina Oprea, and Ronald L Rivest. Flipit: The game of "stealthy takeover". *Journal of Cryptology*, 26(4):655–713, 2013.

[36] Wikipedia. Force protection condition, 5 2015.

[37] Wikipedia. Defcon, 1 2016.

[38] Kara Zaffarano, Joshua Taylor, and Samuel Hamilton. A quantitative framework for moving target defense effectiveness evaluation. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 3–10, New York, NY, USA, 2015. ACM.

[39] Rui Zhuang, Alexandru G. Bardas, Scott A. DeLoach, and Xinming Ou. A theory of cyber attacks: A step towards analyzing mtd systems. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 11–20, New York, NY, USA, 2015. ACM.

[40] Rui Zhuang, Scott A. DeLoach, and Xinming Ou. Towards a theory of moving target defense. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 31–40, New York, NY, USA, 2014. ACM.

[41] Rui Zhuang, Su Zhang, Scott A DeLoach, Xinming Ou, and Anoop Singhal. Simulation-based approaches to studying effectiveness of moving-target network defense. In *National symposium on moving target research*, 2012.

DISTRIBUTION:

1	MS 0359	D. Chavez, LDRD Office, 1911
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories