Dependency Graph Analysis and Moving Target Defense Selection

Jason R. Hamlet
Sandia National Laboratories
Albuquerque, NM
jrhamle@sandia.gov

Christopher C. Lamb
Sandia National Laboratories
Albuquerque, NM
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 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.

CCS Concepts

•**Security and privacy** → **Formal security models;** *Malware and its mitigation;*

Keywords

Dynamic Defense; Moving Target Defense; Cybersecurity

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 naïve 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 cybercrime, they would start to get involved. And they did, as did others, leading us to where we are today [3, 6]. 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 [25] system, as well as the newer FPCON [24] 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.

The primary contribution of this paper is an approach to measure the possible efficacy of a given MTD. Here, we outline a method using dependency analysis to highlight the impacts to all users of a given system, including attackers, that shows the distribution of effort caused by a given defense. After outlining this dependency graph method of impact analysis, we then work through an example to demonstrate how this method could be applied to a deployed, real-world MTD.

2. 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 system 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

This work is being performed under the Laboratory Directed Research and Development Program at Sandia National Laboratories. 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 $DE - AC04 - 94AL85000$.

MTD'16, October 24 2016, Vienna, Austria

ACM ISBN 978-1-4503-4570-5/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2995272.2995277

those measures to represent the performance of the simulated system [26]. 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 [28]. 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 [27]. A hyper-geometric probability distribution model has been proposed for studying the effects of various types of defenses within a computer network [8]. The authors 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 [18].

3. 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 [22]. In this model higher levels are dependent on lower levels, so network layer devices have dependencies on the data-link and physical layers [7]. 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 [29]. Carvalho et al. have proposed a command and control system to appropriately coordinate movements in the system [5]. 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 [4, 9], defense coverage, unpredictability, and timeliness [14], and measures of deterrence, deception, and detectability [16].

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 $f_w^u : \mathbb{R} \to \mathbb{R}, w_k^u = f_w^u\left(c_i^u(k), i = 1 \dots n_u\right)$ where the $c_i^u(k), i = 1 \dots n_u$ are the n_u user costs associated with the edge $k \in E_u$. We similarly define defender labels $w_k^d, k \in E_d$ by $f_w^d : \mathbb{R} \to \mathbb{R}, w_k^d = f_w^d\left(c_i^d(k), i = 1 \dots n_d\right)$ for defender costs $c_i^d, i = 1 \dots n_d$ and adversary labels $w_k^a, k \in E_a$ by $f_w^a : \mathbb{R} \to \mathbb{R}, w_k^a = f_w^a\left(c_i^a(k), i = 1 \dots n_a\right)$ for adversary costs $c_i^a, i = 1 \dots n_a$. 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 from a root node to a terminal node in the user's graph, we define the cost of the path as $f_c^u : \mathbb{R} \to \mathbb{R}, p_j^u = f_c^u\left(w_i^u, i = 1 \ldots n\right)$. 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 [11]. We similarly define the cost of a path 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 in the adversary's graph by $f_c^a : \mathbb{R} \to \mathbb{R}, p_j^a = f_c^a\left(w_i^a, i = 1 \ldots n\right)$ 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 $P_u = \left\{p_{i^u, i=m_u}\right\}, P_d = \left\{p_{i^d, i=m_d}\right\}$ and $P_a = \left\{p_{i^a, i=m_a}\right\}$ 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.

4. 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), a code location randomization technique. It is important to note that we are demonstrating how this dependency analysis technique could be used rather than endorsing any particular set of metrics. While we have attempted to select metrics for this demonstration that are reasonable, we understand that they are not validated in any way. They are used purely for demonstrative purposes.

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 by 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.

4.1 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 [12]. 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.

4.2 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 closer 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 $p = \frac{m-d}{m}$ where m is a scaling factor 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 this is $d = m - pm$ and so the composed cost of the multi-stage dependency is

$$c = m - m \prod_i p_i \tag{1}$$

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

4.3 Demonstration

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. 1 as the distance metric [11].

4.3.1 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 for responding to the defense.

First, we discuss the initial scenario before any defenses are deployed. Considering Figure 1 (see Appendix), 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 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 the user's dependencies are initially assigned zero cost. The adversary's original dependencies are shown by the orange paths in Figure 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 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 to overflow the buffer to inject the malicious

Table 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

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 1. In Table 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 unpredictability, f_m is the frequency of movement, and o_c is the overall cost.

Figure 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, where we have included a 10% change in CPU requirements for 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 [23]. 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. Some of the adversary dependencies, such as learning the reliable location of the return address and modifying a pointer with the address of the 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. 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 [20]. In these

Table 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

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.

In Figure 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 [19]. 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 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 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 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 3.

Individually, both ASLR and DEP can be bypassed relatively easily with well-known techniques, and so they are not particularly effective when used independently [23, 21]. 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 4. Here, we find that because

Table 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

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 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 [10], 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 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 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 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 6.

Finally, we also note that it is possible for an attacker to bypass a protection entirely. For example, an attacker

Table 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

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.

Table 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

5. OVERALL DEPENDENCY ANALYSIS

5.1 Optimization

In section 4.3.1 we demonstrated the use of dependency graphs to analyze a moving target defense. We also ap-

Table 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

plied 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 3 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.

5.2 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 combining the user, attacker, and defender digraphs.

5.2.1 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 [2]. 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.

5.2.2 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 [13]. 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.

5.2.3 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.

5.2.4 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.

Table 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

5.2.5 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 6, the multi-objective optimization described in section 5.1 is trivially solved as a full implementation of ASLR and DEP. The graph analysis approaches of section 5.2 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 1. We begin with edge betweenness centrality. Recall from Section 5.2.1 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 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 are not directly 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 [17, 1, 15]. Since the user's dependency graph has a single path the concept of betweenness centrality is not useful for analyzing it.

Next, we consider the graph efficiency metrics. Recall from Section 5.2.4 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 consists almost entirely of serial connections, we find that node $n38$, which leads to a set of parallel paths 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 ef-

ficiency. 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 5.2.2 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 modifying the execution environment between exploit development or insertion and execution, may be useful.

5.2.6 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. This 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.

6. CONCLUSIONS

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 and data execution prevention. 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.

7. REFERENCES

[1] Barrantes, Elena Gabriela, e. a. Randomized instruction set emulation. In *ACM Transactions on Information and System Security (TISSEC) 8.1* (2005), ACM, pp. 3–40.

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

[3] Broadhurst, R., Grabosky, P., Alazab, M., Bouhours, B., and Chon, S. 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 (2014), 1–20.

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

[5] Carvalho, M. M., Eskridge, T. C., Bunch, L., Bradshaw, J. M., Dalton, A., Feltovich, P., Lott, J., and Kidwell, D. 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* (2013), ACM, p. 38.

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

[7] Crosby, S., Carvalho, M., and Kidwell, D. 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* (2013), ACM, p. 36.

[8] Crouse, M., Prosser, B., and Fulp, E. W. Probabilistic performance analysis of moving target and deception reconnaissance defenses. In *Proceedings of the Second ACM Workshop on Moving Target Defense* (New York, NY, USA, 2015), MTD '15, ACM, pp. 21–29.

[9] Cybenko, G., and Hughes, J. No free lunch in cyber security. In *Proceedings of the First ACM Workshop on Moving Target Defense* (New York, NY, USA, 2014), MTD '14, ACM, pp. 1–12.

[10] Dai Zovi, D. Practical return-oriented programming. *SOURCE Boston* (2010).

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

[12] Duggan, D. P. Generic threat profiles. Tech. Rep. SAND2005-5411, Sandia National Laboratories, 2005.

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

[14] Hobson, T., Okhravi, H., Bigelow, D., Rudd, R., and Streilein, W. On the challenges of effective movement. In *Proceedings of the First ACM Workshop on Moving Target Defense* (New York, NY, USA, 2014), MTD '14, ACM, pp. 41–50.

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

[16] Jafarian, J. H. H., Al-Shaer, E., and Duan, Q. Spatio-temporal address mutation for proactive cyber agility against sophisticated attackers. In *Proceedings of the First ACM*

Workshop on Moving Target Defense (New York, NY, USA, 2014), MTD '14, ACM, pp. 69–78.

[17] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security.* (2003), ACM.

[18] Miehling, E., Rasouli, M., and Teneketzis, D. Optimal defense policies for partially observable spreading processes on bayesian attack graphs. In *Proceedings of the Second ACM Workshop on Moving Target Defense* (New York, NY, USA, 2015), MTD '15, ACM, pp. 67–76.

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

[20] Rohlf, C., and Ivnitskiy, Y. Attacking clientside jit compilers. *Black Hat USA* (2011).

[21] Sotirov, A. Bypassing memory protections: The future of exploitation. In *USENIX Security* (2009).

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

[23] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 48–62.

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

[25] Wikipedia. Defcon, 1 2016.

[26] Zaffarano, K., Taylor, J., and Hamilton, S. A quantitative framework for moving target defense effectiveness evaluation. In *Proceedings of the Second ACM Workshop on Moving Target Defense* (New York, NY, USA, 2015), MTD '15, ACM, pp. 3–10.

[27] Zhuang, R., Bardas, A. G., DeLoach, S. A., and Ou, X. A theory of cyber attacks: A step towards analyzing mtd systems. In *Proceedings of the Second ACM Workshop on Moving Target Defense* (New York, NY, USA, 2015), MTD '15, ACM, pp. 11–20.

[28] Zhuang, R., DeLoach, S. A., and Ou, X. Towards a theory of moving target defense. In *Proceedings of the First ACM Workshop on Moving Target Defense* (New York, NY, USA, 2014), MTD '14, ACM, pp. 31–40.

[29] Zhuang, R., Zhang, S., DeLoach, S. A., Ou, X., and Singhal, A. Simulation-based approaches to studying effectiveness of moving-target network defense. In *National symposium on moving target research* (2012).

APPENDIX

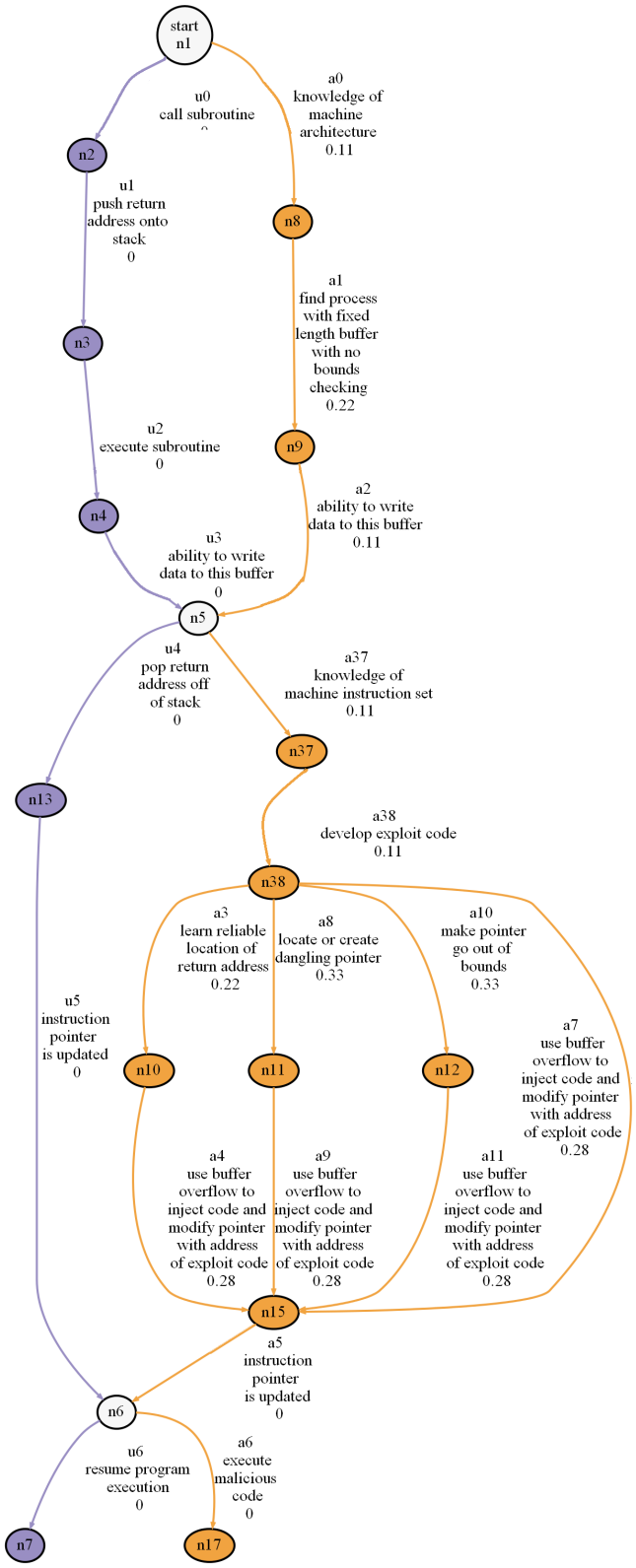

Figure 1: Dependency graphs for users and adversaries prior to applying any defenses.

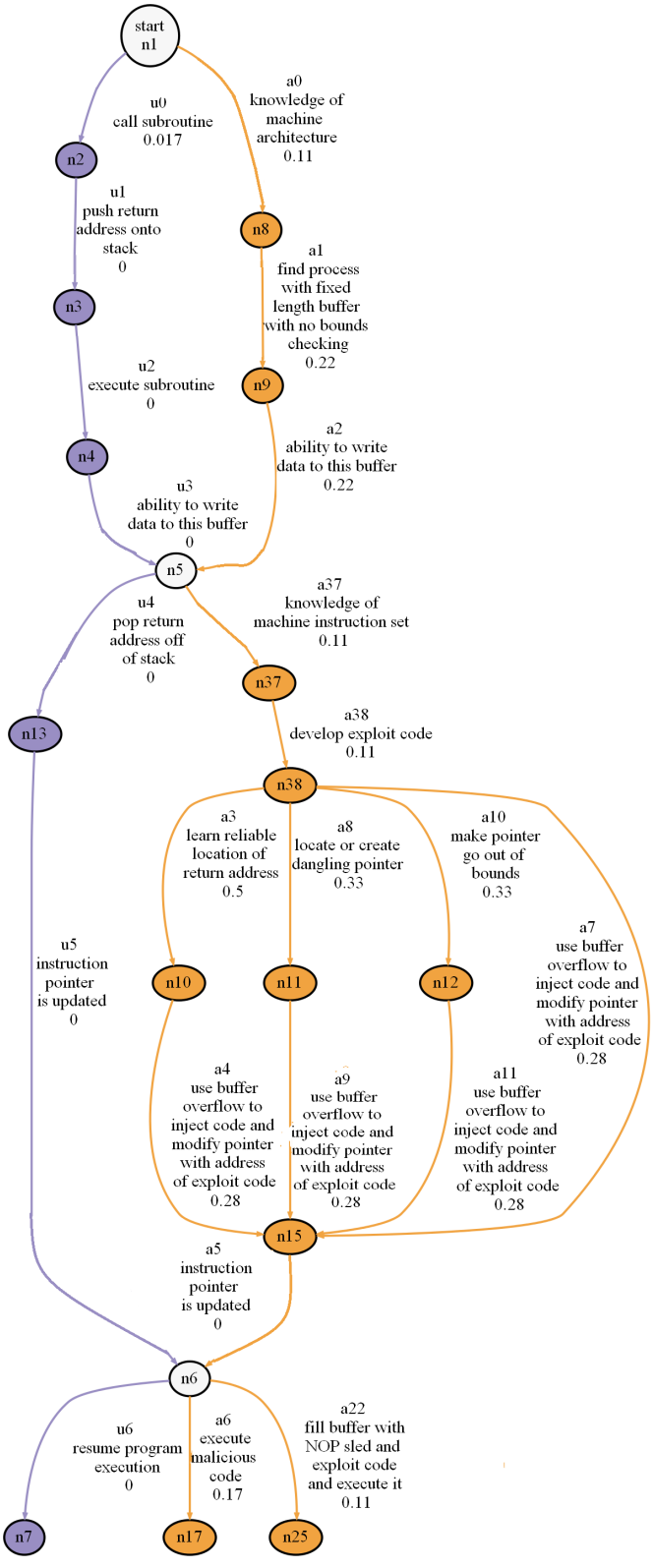

Figure 2: Updated dependencies after applying address space layout randomization. Adversaries no longer have reliable knowledge of the location of injected code within memory, making NOP-sled type attacks the most likely.

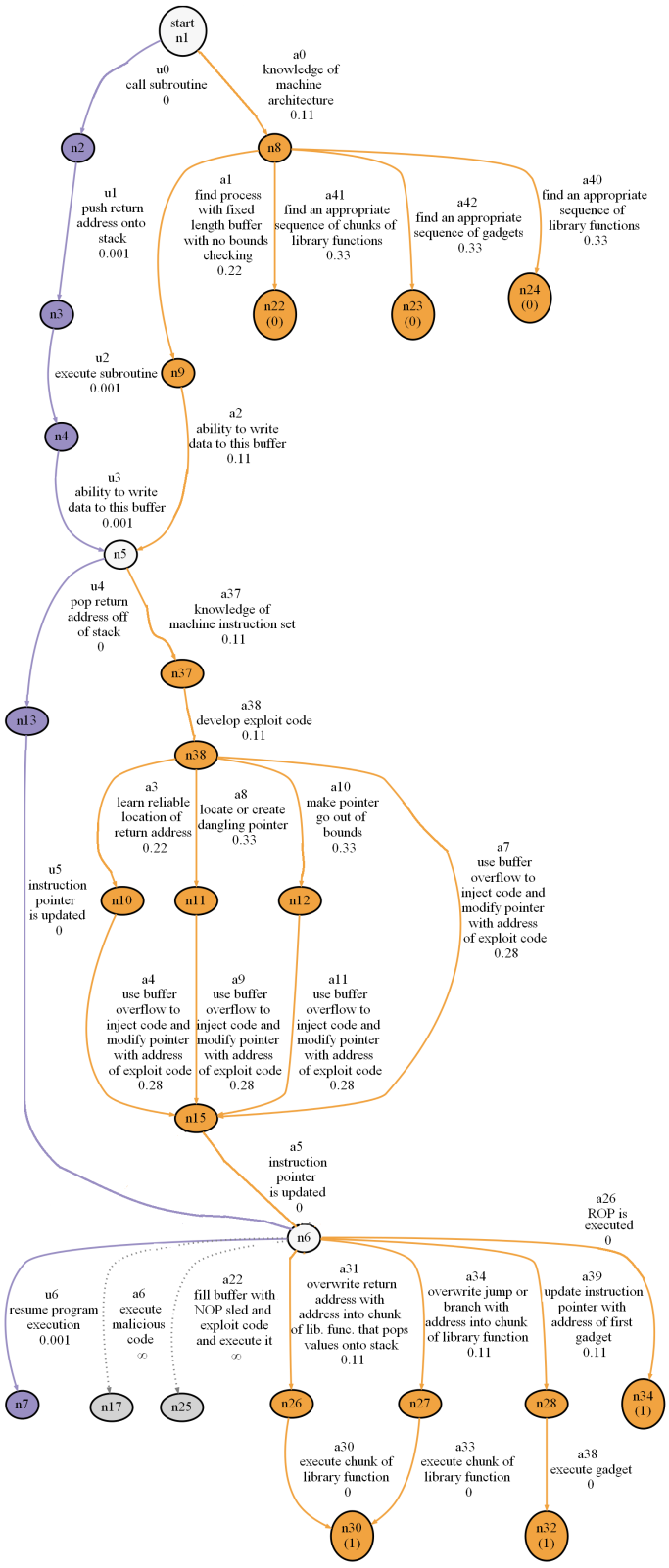

Figure 3: After applying data execution prevention adversaries can no longer reliably execute injected code, and instead are likely to use return-oriented programming techniques.

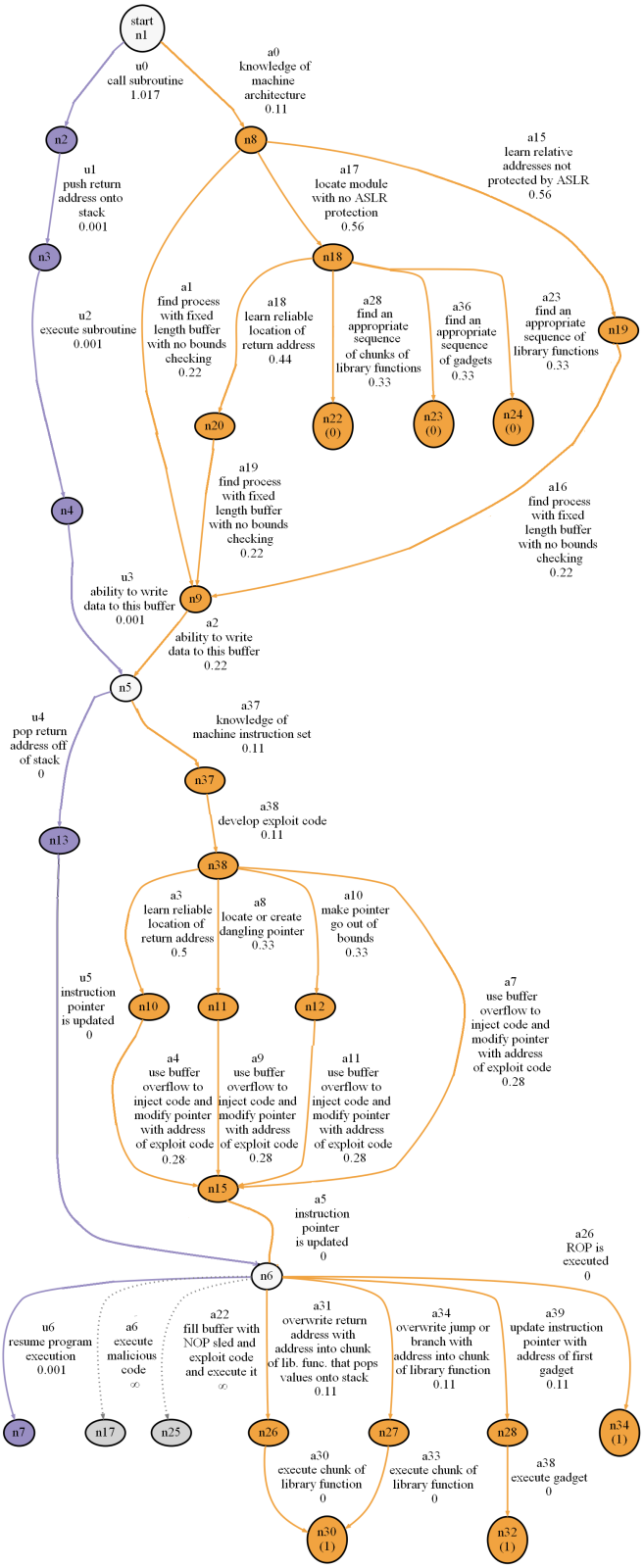

Figure 4: After implementing DEP and applying ASLR to some executables the adversary is still likely to use return-oriented programming techniques, but will now need to build the malicious code only from modules that are not protected by ASLR.

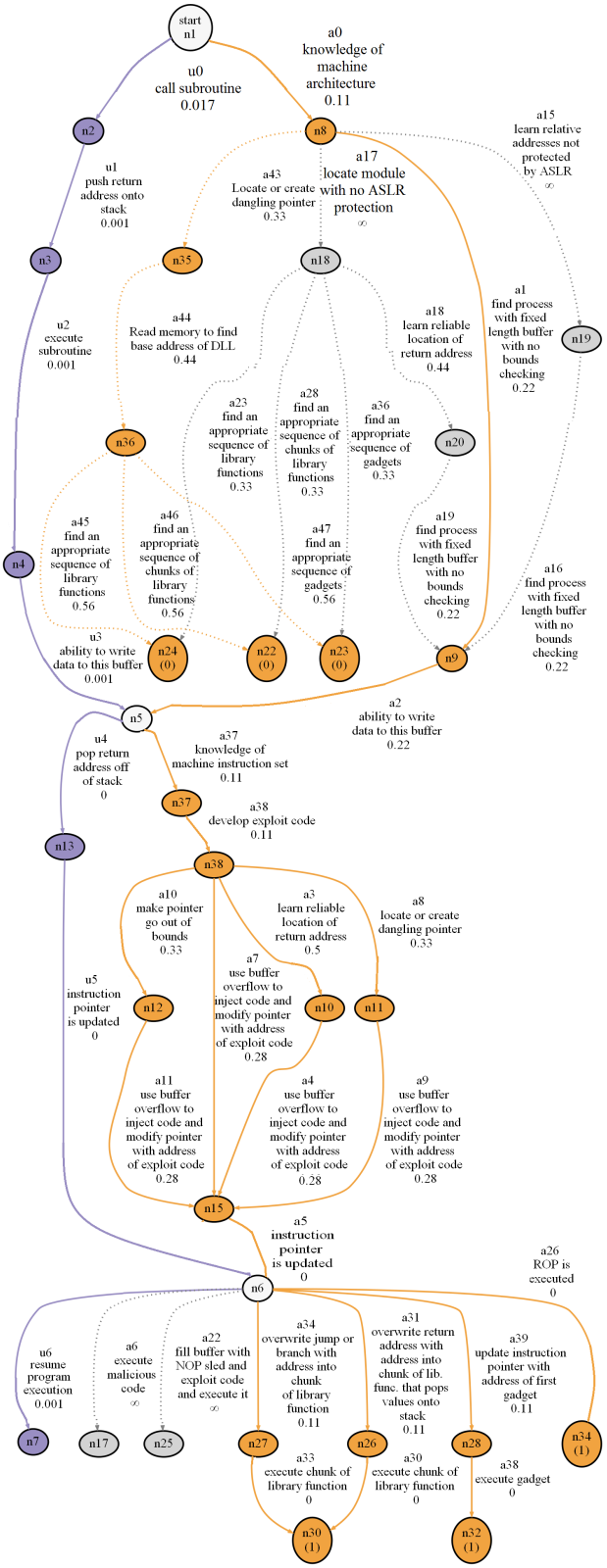

Figure 5: After implementing DEP and protecting all modules with ASLR, the adversary must exploit a memory disclosure vulnerability to learn the base address of a module, and then dynamically construct malicious ROP code.