# Misery Digraphs: Delaying Intrusion Attacks in Obscure Clouds

Hussain M. J. Almohri, *Member, IEEE*, Layne T. Watson, *Life Fellow, IEEE*, and David Evans, *Member, IEEE*

***Abstract*— When remote command injection attacks succeed at the entry points of a cloud (servers exposed to the outside Internet), attackers targeting a specific asset in the cloud will pursue further exploration to find their targets. Attack targets, such as database servers, are often running on separate machines, forcing an extra step for a successful attack. However, compromising two or three machines is all an attacker needs to reach an isolated database through a simple attack path. The goal of this paper is to investigate the possibility of frustrating attackers by constructing a cloud network architecture that hides the path to a target asset in the network, utilizing multiple *moving* decoy virtual machines and confusing firewall configurations. A deceiving cloud network architecture can significantly delay attacks (by stretching the attack path from a handful of steps to thousands), providing time for system administrators to intervene and resolve the intrusion. This paper introduces the concept of *misery digraphs*, which provide a theoretical foundation for creating intrusion deception in clouds. This paper describes the necessary steps to convert a cloud to one that includes a misery digraph, and evaluates the feasibility and effectiveness of using the approach with Amazon Web Services. Our simulation results demonstrate that for a cloud implementing misery digraphs with a simple attack path of length five, there is a 91% probability that an attack requires at least 1000 steps to reach the target.**

***Index Terms*— Network security, security management, data security, tree graphs.**

## I. Introduction

REMOTE code execution attacks [28] exploit vulnerable network services for transferring malicious commands to the host's operating system. In vulnerable applications, attackers often exploit unfiltered parameters [26], such as the ones passed to require or exec in a PHP script to execute a command that could be powerful enough to modify authentication policies, creating permanent unauthorized access to the host. Besides vulnerable applications, widely used network services potentially increase the vulnerable surface. For example, even though key-based SSH authentication is a well-established practice, system administrators can still choose to enable both key-based and password-based authentication, enabling attacks that target passwords. An analysis of Amazon EC2 instances revealed attackers' interest in brute force SSH attacks [1].

Remote code execution attacks can be harmful when an attacker is motivated to reach a specific target *within* a cloud-based virtual network. To succeed, the attacker searches for a vulnerable entry point in the virtual network's external-facing hosts and uses a compromised edge host to launch an attack on the target asset. As target assets are isolated from public Internet gateways, the attacker must repeat the attack process by finding and exploiting vulnerable machines that have direct access to the target. In common cloud network architectures, successful attacks require only a few steps (compromising two or three hosts). The goal of this work is to increase the number of steps needed and make each step more difficult for the attacker.

We investigate a pure architectural solution, utilizing unique services provided by cloud computing platforms, to mitigate remote code execution attacks. Our approach is to continuously change the structure of the virtual network. This confusing architecture provides parallel support for the intrusion detection systems, transforming the firewall rules of the cloud into a complementary line of defense. Our goal is not necessarily to detect or eliminate the attack, but to confuse and deceive attackers in ways that impose severe delays on the attack process.

Several previous works have advocated for creating deception or using moving target strategies to combat intrusions either in a virtual network or in a physical network. Several works focused on physical networks, developing a number of deception strategies based on network overlays, proxying, and secret IP addresses [4], [16], [23], [27]. Other proposals progressed closer to the present goals by leveraging the elasticity provided by cloud computing platforms to distract distributed denial of service attacks (DDoS) [8], [17], [34]. For example, to prevent DDoS attacks on specific targets, Jia *et al.* [17] proposed creating replicas on the fly and assigning network traffic to new replicas. Although this work, as well as Brezeczko *et al.*'s [5], provide innovative techniques to utilize cloud resources against remote attacks, they do not consider attacks that continue to intrude into the network and search for isolated targets. This paper addresses the question of how to combine attack deception and a moving target strategy. In particular, we present and analyze a mathematical cloud architectural model for significantly delaying intrusion attacks

Manuscript received June 9, 2017; revised October 8, 2017; accepted November 23, 2017. Date of publication December 4, 2017; date of current version February 7, 2018. This work was supported by Kuwait University under Project RQ02/15. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Vrizlynn L. L. Thing. *(Corresponding author: Hussain M. J. Almohri.)*

H. M. J. Almohri is with the Department of Computer Science, Kuwait University, Kuwait, and also with the University of Virginia, Charlottesville, VA 22903 USA (e-mail: almohri@ieee.org).

L. T. Watson is with the Departments of Computer Science, Mathematics, and Aerospace and Ocean Engineering, Virginia Tech, Blacksburg, VA 24061 USA (e-mail: ltw@ieee.org).

D. Evans is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903 USA (e-mail: evans@virginia.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2017.2779436

that go beyond DDoS and propagate through a cloud-based virtual network aiming to compromise a target asset.

The key insight here is to introduce myriad dynamically changing redundant attack paths that hide the real attack path from an attacker and create confusion about the steps that an attacker must take to reach a target database server. We use a cloud's firewall rules as the basis for an abstract model of the cloud's security architecture. Then, we transform the resulting model into a set of *misery digraphs*, graph theoretic models for confabulating the real attack paths. This transformation maintains the original intention of the cloud designers, does not impose modifications on the core application logic, and incurs a prohibitive delay to attacks while causing minimal delay to legitimate application traffic. A virtual network containing misery digraphs forces a targeted attack to traverse a longer path to a target, requiring multiple blind decisions to finding the true path to target. Misery digraphs possess a symmetric structure for confusing attackers, while neutrally multicasting the legitimate traffic throughout the virtual network. The dynamic behavior of attack paths in the misery digraph causes a continuously changing virtual network structure that repeatedly wastes an attacker's efforts by modifying and relocating redundant machines that are necessary to compromise before reaching the target.

Transforming basic virtual networks into misery digraphs faces several challenges. First, the main challenge is that misery digraphs should provide consistent and complex paths towards target assets without leaking side-channel information that would provide attackers with path-pruning opportunities. Second, misery digraphs must require minimal modifications to the underlying application. Third, the structure of misery digraphs must have financial efficiency, measured in the increased hourly cost of the cloud. Section II provides background on cloud applications and details on these challenges. Section III describes the design of misery digraphs and shows how it addresses these changes for state-of-the-art cloud architectures. Section IV presents the algorithms for constructing misery digraphs, along with a cost analysis leading to a formula to estimate the cost of our defense. Section V provides a security analysis and demonstrates the various scenarios in which misery digraphs are useful and also discusses limitations of our approach. Section VI evaluates the effectiveness of misery digraphs using an extensive simulation, showing that a minimal misery digraph can extend attack paths to thousands of steps. Our evaluation also considers using misery digraphs to protect an Amazon Web Services web application, demonstrating that misery digraphs can be applied in practice with reasonable cost. The main contributions of the work are:

1) developing a rigorous graph-theoretic model for creating deception in a cloud network of virtual machines (Section III-A),
2) designing a moving target strategy in the model where the true path to a target machine is continuously moved around the network (Section III-B),
3) presenting algorithms for automatically generating confusing clouds using existing cloud settings (consistent with practical cloud computing models) (Section IV),
4) analyzing the results of a deep simulation of an attack on a cloud with a misery digraph (Section V), and
5) demonstrating the practicality of the approach and a concrete cost analysis by creating a prototype AWS virtual network containing misery digraphs (Section VI).

Misery digraphs achieve a high level of deception. Simulating an attacker, our results demonstrate that with a reasonably fast changing misery digraph, for a network of two machines (an application and a database server), there is a 91% probability that the attacker must compromise 1000 virtual machines before successfully connecting to the target server.

The presented model is tested using the core and stable technology provided by Amazon Web Services (AWS). Misery digraphs require customized virtual machines, virtual internal networks, Internet gateways, and intermachine firewall rules. Thus, the model is consistent with cloud computing practice and does not assume abstract computing or special security services available in major cloud providers.

## II. Overview of the Problem

This section provides background on current cloud computing platforms, and motivates the requirements for our design.

### A. Background

Cloud providers, such as Amazon Web Services (AWS), facilitate virtual private clouds, networks of virtual instances, and virtual subnets with features resembling physical networks along with other unique features such as security groups, software-based load balancers, and elastic IP addresses. With a fresh virtual private cloud, a system administrator chooses from a range of virtual instance types (virtual machines that may or may not map to physical machines), managed database instances, or specialized instances (such as a machine learning service).

One of the sixteen reference cloud architectures proposed by AWS[1] is the Web Application Hosting architecture depicted in Figure 1. This architecture is an example of a possible target for remote intruders who may be motivated to attack web applications and gain unauthorized access to their data. The Web Application Hosting architecture employs software-based load balancers that route requests to a basket of IP addresses and provide no other public interface. The cloud user—the system administrator that manages the cloud—creates a number of virtual instances to execute the application logic. The cloud user has to use SSH servers on the virtual instances since there is no physical access to the machines. By connecting to these *entry point* instances with SSH, the cloud user can walk through the network and manage internal instances. In the architecture of Figure 1, only the web servers face the external Internet, the application servers process requests and mediate access to the database server, which contains the sensitive data.

### B. Motivation

Equifax data breach and the 2014 Target data breach are examples of vulnerabilities that allowed for arbitrary code

[1] https://aws.amazon.com/architecture/

execution by exploiting web servers and executing commands on them. In these incidents, attackers did not directly query the data through vulnerable applications, for example by using SQL injection attacks. Instead, in a more complex attack process, a remote code execution vulnerability allowed for some limited exploitation of a server that was facing the Internet. Then, the exploited entry point server, running a vulnerable version of the web server software allowed access to the attacker, which was subsequently used to gain access to other machines that would have direct access to data.

The Equifax data breach was reported [9] to have been a consequence of a vulnerablity in Apache Struts 2.1.2 and before 2.3.34 [10], which allowed for arbitrary code execution. In this attack, a decerialization flaw allows for unsanitized data to be converted into Java objects. Using these vulnerabilities, the attacker aims for executing code within the program's context, eventually leading to executing commands on the target system. Moreover, in the 2014 Target data breach, the attackers used island hoping, compromising and exploiting multiple machines to reach internal Target servers [7]. Unlike many SQL injection attacks that depend only on vulnerable applications, usually through unfiltered POST or GET requests, island hopping attacks use remote code execution vulnerabilities to gain access to intermediate machines and hop through the network to reach their targets.

The main ingredient of these attacks is the requirement and ability of the attacker to quickly move across the network by connecting to entry point and intermediate machines to find a direct access path to a valuable target. Specifically, a successful attack is bound to a set of conditions:

1) Hijacking normal HTTP requests from clients to inject malicious queries without compromising the server is not possible or will only yield limited results, making the attack impractical.
2) The attack targets valuable data or application logic that are not directly accessible from the entry point machines, which face the Internet. Therefore, attacks cannot commit in a single step.
3) The exact structure of the internal topology of the cloud-based virtual network isn't visible to the attacker. Thus, the attacker has to incorporate a search strategy to reconstruct an abstract image of the target network.

Throughout this article, we use an example cloud-based virtual network architecture that includes a valuable target to which the attackers wish to gain direct access. Sections II-C and II-D present the details of the problem and the capabilities of attackers and defenders.

### C. Problem Statement

Our goal is to *delay* the time to success of remote network attacks motivated to compromise the source of data within a cloud application. The attacker's ultimate goal is either to corrupt or query the target database. To gain access to a database server, the attacker must compromise an entry point in the cloud and propagate through the cloud. The benefit of delaying an attacker in a confusing cloud architecture is providing a larger response time window when an intrusion is detected.

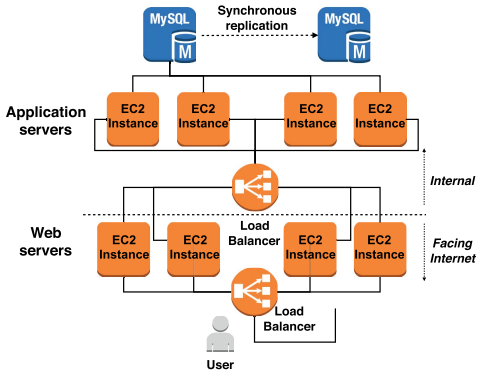



Fig. 1. An example Web Application Hosting architecture recommended by AWS. The architecture includes a cluster of EC2 instances (virtual machines in AWS) hosting the application, isolating a cluster of database machines that are only internally accessible. This example architecture has four EC2 instances as web servers, four EC2 instances as application servers, and a main and a replicated database server. DB-SYNC is used to synchronize the two database instances.[2]


Fig. 2. A chain of steps that an attacker must take towards a target database server, requiring two steps to reach the target database.

A long enough delay may cause enough frustration or cost to the attacker to be sufficient to thwart the attack. Note that providing the architectural support for delaying the attacks is an orthogonal problem to detecting the intrusion within the cloud and utilizing the delayed and complicated attack path for preventing unauthorized access to the database server. It is also orthogonal and complementary to the goal of eliminating the vulnerabilities in the first place. The best defense would, of course, be to remove those vulnerabilities, leaving the attacker with no starting point for the attack. But, eliminating vulnerabilities from complex applications remains an elusive goal, motivating our work on mitigating their exploitation.

Figure 2 demonstrates one possible chain of steps to compromise the database server of Figure 1. In this chain, the attacker will compromise an EC2 instance that has an *elastic IP* address, which is a public IP address service provided by Amazon Web Services. Next, with shell access to the compromised machine, the attacker will either use a stored credential to connect to the next EC2 instance or use another vulnerability to take control of the next EC2 instance. Finally, once the attacker controls the EC2 instance with direct access to the database, the attacker can access and manipulate the database directly.

The architecture of Figure 1 is modeled in a graph of firewall rules extracted from within a cloud, capturing the connectivity structure among virtual instances. Two machines are connected if there is a network path between the two machines and the firewall rules allow traffic to pass through the path. In AWS, the firewall rules are explicitly defined in

[2]Figures 1 and 2 are created using AWS Simple Icons: https://aws.amazon.com/architecture/icons/

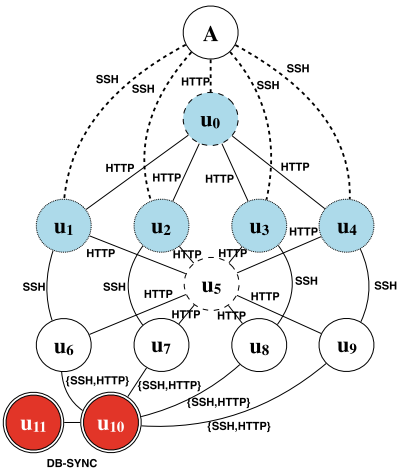



Fig. 3. A connectivity digraph for Figure 1. Each vertex represents a machine in Figure 1 and each edge corresponds to the connectivity of the machines. Internet traffic (represented by vertex $A$ and dotted lines) flows to the load balancer $u_0$, which forwards HTTP requests to $u_1, \ldots, u_4$. $u_0$ is a HTTP entry point while $u_1, \ldots, u_4$ are SSH entry points. Load balancers are dashed, entry points are in blue, and target machines have double circles and are in red. The edge labels capture the protocols used for the connection. Access to $u_{10}$ is assumed to be through HTTP for the hosted application. Edges labelled as $\{SSH, HTTP\}$ indicate two services allowed to pass through.

security groups. Formally, machine connectivity is defined by a labelled digraph.

*Definition 1:* A *connectivity-labelled digraph*, $L = (V, A, \ell)$, is a digraph where $v \in V$ represents a vertex corresponding to an instance in the cloud, $A$ is a set of $(u, v)$ pairs representing directed edges from $u$ to $v$ in the digraph, and $\ell : A \rightarrow 2^S$ gives the set of possible connections in $A$. For some service protocol $s \in S$, $s \in \ell(u, v)$ means $u$ can establish connections to $v$ and $v$ accepts connections from $u$ on $s$.

Figure 3 shows an example connectivity digraph $G$ for the cloud of Figure 1. In this digraph, machines from Figure 3 including load balancers, web servers, application servers, and database servers are all represented by vertices. The undirected edges represent inbound and outbound traffic (arcs) between two vertices as set by firewall rules. These rules correspond to the lines connecting the machines in Figure 3. The main network traffic passes through the load balancer, $u_0$. However, $u_0$ is not the only entry point in the digraph. Vertices $u_1, \ldots, u_4$ also represent entry points as they are open to SSH traffic. System administrators open access to these vertices because $u_1, \ldots, u_4$ are virtual machines with no physical access.

A successful attack in $G$ is capable of compromising one of $u_1, \ldots, u_4$ and propagating through a path towards $u_{11}$ or $u_{12}$ depending on the attack's motivation. Although $u_0$ is also an entry point, it plays no role in facilitating the attack except for forwarding the requests to $u_1, \ldots, u_4$. Similarly, the load balancer, $u_5$, acts as a forwarder enabling a malicious request to travel through $u_6, \ldots, u_9$.

The cloud of Figure 3 allows an attack path with only four steps to reach the target. For example,

$$\pi = \mathcal{A} \xrightarrow{\text{SSH}} u_1 \xrightarrow{\text{SSH}} u_6 \xrightarrow{\text{HTTP}} u_{10} \xrightarrow{\text{DB-SYNC}} u_{11}.$$

Even with a significantly larger digraph, with the same architecture, the size of the shortest attack path remains unchanged, and the more possible paths to the target, the more opportunities for the attacker. The intent of this work is to ensure there are no short paths to reach the target available to the adversary.

### D. Threat Model

We assume a trusted and uncompromised cloud computing platform, and focus on protecting an application running on that cloud from a sophisticated and motivated adversary who aims to gain unauthorized access to a targeted data asset within the cloud.

We assume a powerful attacker with the ability to compromise hosts along a path (such as $\pi$ in Section II-D) through the application. Such an attacker can find an entry point machine, perhaps by using an IP scan of a range of the hosting cloud, and compromise that host to launch attacks on other hosts with the goal of reaching the target. We assume other hosts also have vulnerabilities that can be exploited by the adversary.

We assume the attacker has no control over the cloud settings that determine the number and the types of instances, the network structure, and global firewall rules that control access to the virtual private cloud. An attacker who can compromise the cloud provider or the application owner's configuration access is beyond the scope of this work. Since the attacker has no control over the cloud, we assume that the attacker's knowledge about the connectivity structure of the internal network in the cloud is not complete. Specifically, the attacker does not know the number of virtual machines and their security groups within the internal network. This information can only be incrementally revealed by attacking the virtual network. When the cloud is first created, the attacker only knows that for some $i$, $u_i \in V$ is vulnerable and open to Internet traffic, and a target instance $t \in V$, accessible from $u_i$, exists.

*1) Defender's Capabilities:* The defender's goal is to create a cloud network of virtual machines that preserves the original functionality with minimal additional cost, while frustrating attacks. The defender achieves this by having full control on the cloud settings and all the virtual machines. In a trusted cloud computing platform, only the defender can control the security groups, providing or revoking access to individual virtual machines. For example, in AWS the cloud user adds or removes machines, changes access control among the machines, and has full access to all virtual machines. The attacker cannot tamper with virtual machine access control rules from within a compromised virtual machine (for example, by changing iptable rules) as the access control rules will be firmly overridden by the cloud.

## III. Design of Misery Digraphs

The strategy for delaying a remote command injection attack is (i) to create a large network of decoy virtual machines to confuse the attacker, and (ii) dynamically relocate and modify the decoys to waste the attacker's resources and frustrate the attacker. Increasing attack complexity and duration starts with expanding an initial connectivity digraph of an existing virtual network into one containing *misery digraphs*.

TABLE I
A TABLE OF SYMBOLS USED IN DEFINITION 2

| Symbol | Description |
|---|---|
| $L$ | A connectivity-labelled digraph |
| $\pi$ and $\pi^R$ | A path and its reverse in $L$ |
| $G$ | An expanded path from $L$, which includes extra vertices |
| $p$ and $p^R$ | A path and its reverse in $G$ |
| $G^*$ | A misery digraph with $b$ paths, including $G$ |
| $k$ | Depth of a misery digraph $G^*$ |

### A. Defining Misery Digraphs

A misery digraph contains the original virtual network combined with additional deceiving structure. In a misery digraph, at each point in time, only a single path has bidirectional access to the target server. As a random function of time, uniformly selected pairs of decoy virtual machines are replaced and switch positions within the misery digraph. As a result, misery digraphs change the true path to the target, disabling the attacker from learning the structure of the virtual network.

We first define a generic misery digraph, building on our definition of a connectivity-labelled digraph from Section II-C. Then, we define canonical misery digraphs, which instantiate the generic misery digraphs, and discuss the main two properties of misery digraphs, which is a periodic relocation of machines and hiding the true path towards the target. A symbol reference is provided in Table I.

*Definition 2:* Let $L$ be a connectivity-labelled digraph with a path $\pi = (u_1, u_{i_1}, \ldots, u_{i_m}, u_{k+1})$ connecting an entry vertex $u_1$ to a target vertex $u_{k+1}$ and containing the reverse path $\pi^R = (u_{k+1}, u_{i_m}, \ldots, u_{i_1}, u_1)$. $\pi$ is enlarged to a path of length $k$ in the digraph

$$G = \left(\{u_i\}_{i=1}^{k+1}, \left\{(u_i, u_{i+1}), (u_{i+1}, u_i)\right\}_{i=1}^{k}\right)$$

consisting of a single path $p = (u_1, \ldots, u_{k+1})$ from the entry vertex $u_1$ to the target vertex $u_{k+1}$, and the reverse path $p^R = (u_{k+1}, \ldots, u_1)$. A *misery digraph*, $G^*$, for $\pi$ contains $G$ as a subdigraph, exactly $b$ paths $q^{(j)} = \left(u_1, v_2^{(j)}, \ldots, v_k^{(j)}\right)$, $j = 1, \ldots, b$, of maximal length $k-1$, as well as the arcs $\left(u_{k+1}, v_k^{(j)}\right)$ and reverse paths $\left(q^{(j)}\right)^R$, $j = 1, \ldots, b$. The paths $q^{(j)}$, $j = 1, \ldots, b$, mirror $p$ in that id $u_\ell =$ id $v_\ell^{(j)}$ (in degrees) for $\ell = 2, \ldots, k$ and od $u_\ell =$ od $v_\ell^{(j)}$ (out degrees) for $\ell = 2, \ldots, k-1$. Note that $G^*$ has depth $k$, some of the $v_\ell^{(j)}$ may equal $u_\ell$, and $G^*$ can be constructed in many different ways.

Two example misery digraphs are created for the virtual network in Figure 1 and depicted in Figure 4. The connectivity digraph of Figure 3 can have multiple misery digraphs, one for each path to target, which can be constructed in various ways. In this scenario, two original paths to $u_{10}$ (through SSH and HTTP) are replicated in completely redundant paths. The example shown in Figure 4 captures one possible format of misery digraphs created for a path that uses the SSH service (Figure 4-A) and a similar path that uses the HTTP service (Figure 4-B). In both misery digraphs A and B, only $u_{15}$ sends outbound requests to $u_{10}$. There are three other alternative misery digraphs with a different vertex that has direct SSH or HTTP access to $u_{10}$.

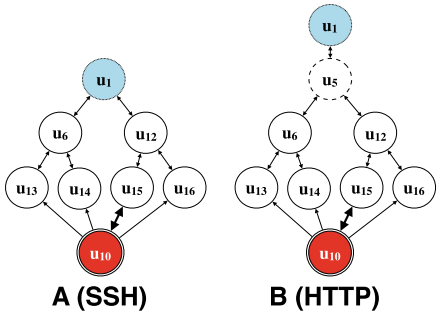



Fig. 4. Two misery digraphs with extra paths and vertices to confuse the attacker about the true path to the target $u_{10}$. The misery digraph A is for SSH and the misery digraph B is for HTTP. To prevent side-channel information, all requests from the entry point are multicast to all paths. Misery digraphs A and B change in time by changing the vertex that has an outgoing arc towards $u_{10}$.

### B. Canonical Misery Digraphs

Misery digraphs can take many forms and produce strategies with various implications. Our goal is to find designs that maximize the cost for the attacker relative to the additional cost for the application owner. These requirements drive our strategy:

1) A misery digraph should not include vertical shortcuts. That is, a misery digraph should not include arcs that lead to pruning entire subgraphs.
2) A misery digraph cannot connect the target server to more than one vertex in the entire digraph. Violating this requirement will make the misery digraph easier to traverse.
3) Target servers (known to the cloud user) should be pushed to the deepest layer in the graph, making them only accessible by paths of at least length $k$.

Minimally fulfilling the requirements above are *canonical misery digraphs*:

*Definition 3:* The *canonical misery digraph*, $\tilde{G}^*$, is a layered digraph with $d+1$ layers. Layer 1 contains only the entry point $a$, layer $d+1$ contains only the target point $t$, and the underlying undirected graph of layers 1, 2, ..., $d$ is a complete balanced $k$-ary tree rooted at $a$, with each edge $\{r, s\}$ corresponding to arcs $(r, s)$ and $(s, r)$ in $\tilde{G}^*$. For each leaf node $w$ in this $k$-ary tree (each node $w$ at level $d$) there is an arc $(t, w)$ in $\tilde{G}^*$, and exactly one arc $(w, t)$ from level $d$ to $t$ in $\tilde{G}^*$.

Canonical misery digraphs contain $k$-ary trees that are balanced and complete, giving the attacker no clue for preferring one path over another. As the attacker traverses the graph, the structure of the digraph only reveals alternative paths that all appear *equivalent* to the attacker. Since the target is also moving, between two points in time the true path towards the target changes, making some of the attacker's discoveries obsolete.

### C. Relocating the Decoys

Definition 2 defines the structure of misery digraphs, which provides a platform for deceiving intruders. For an increased deception in misery digraph, two mechanisms are introduced.

First, misery digraphs change in time, moving the true path to target and *resetting* decoy machines using a random process. Second, misery digraphs hide the true path to target by replicating the traffic towards it. In the remainder of this section, we first present the relocation process for misery digraphs. In Section III-D, we present a method for hiding the true path to target.

Relocating machines in the network involves a random *relocation process*, which interchanges two pairs of vertices within a single layer in a misery digraph. As a random function of time, the current active arc among

$$\left\{\left(u_k, u_{k+1}\right), \left(v_k^{(1)}, u_{k+1}\right), \ldots, \left(v_k^{(b)}, u_{k+1}\right)\right\}$$

is interchanged for a different arc to the target vertex $u_{k+1}$. This ensures that access to the target machine is not static and changes in time. For random $1 \leq i < j \leq b$ and $2 \leq m < k$ the pair of arcs $\left(v_m^{(i)}, v_{m+1}^{(i)}\right)$, $\left(v_m^{(j)}, v_{m+1}^{(j)}\right)$ is replaced by the pair $\left(v_m^{(i)}, v_{m+1}^{(j)}\right)$, $\left(v_m^{(j)}, v_{m+1}^{(i)}\right)$, which randomizes $G^*$. The randomness ensures that at each point in time two entire paths in the digraph are modified by replacing the chosen machines with new virtual machine images and switching their positions in the digraph. Even if the attacker had already compromised a large portion of the selected paths, the attacker's effort is lost.

For example, misery digraphs A and B of Figure 4 change in time to create a moving target. Randomly chosen pairs of edges must be relocated as a function of time, dynamically modifying the paths to target, and refreshing the corresponding decoy virtual machines. $(u_6, u_{14})$ may be chosen to switch with $(u_{12}, u_{15})$, resulting in a lost attack effort, if the attacker had chosen a path containing any of the four nodes.

In a real cloud-based virtual network, relocating the decoys takes place by modifying the cloud's firewall rules that define the accessibility of machines. For example, in Amazon Web Services, a security group of machines defines firewall rules and controls the network reachability of member machines based on the protocol and the port number. As discussed later in Section VI-B, by dynamically modifying security groups, machines can change location in the misery digraphs. Also, another requirement of the relocation process is to reset machines that were relocated. Machines are shutdown using remote APIs available to the cloud owner, and are replaced with new machines using a diversified system configuration (for example, running a different operating system).

### D. Hiding the Path to Target

The effectiveness of misery digraphs depends on the attacker not being able to distinguish correct guesses for the next host from incorrect ones. To maintain an exponential advantage over the attacker, it is important that the attacker has no way to determine if the attack is on the right path until that path has been followed all the way to the target.

The attacker is assumed to have full access to each compromised host, so can fully observe (and alter) that machine's behavior and the flow of requests. Hence, it is important that the attacker who has fully compromised a node at one layer, can only determine which nodes at the next layer are connected to the compromised node, without learning if the compromised node is on the real path. To resolve this potential problem, at each layer of the misery digraph a user's request is multicast to all nodes in the successive layer. That is, each decoy forwards the requests to all others it is connected to in the following layer. On the way back from the target server, all data responses must also be sent back along all paths connected to the target server. For example, in Figure 4 only the arc ($u_{15}$,$u_{10}$) carries actual HTTP requests to the target vertex $u_{10}$, but the decoy nodes all send the same requests. Decoys must be indistinguishable from path nodes, so need to fully duplicate all the computation and communication that would be done on the actual path.

We assume that all request traffic go along a path from the entry points to the target virtual machine from which the responses are sent back. Internal decoy virtual machines only generate responses to requests from the previous layer. Decoys do not modify the responses and do not maintain internal states. Also, it is assumed that the application does not require maintaining internal states throughout the network.

### E. Handling Fault Tolerance

Major cloud providers include load balancer services and recommend architectures that use replicas of web, application, or database servers. Misery digraphs are designed to integrate with load balancers as demanded by the architecture.

As shown in Figure 3, the load balancer $u_0$ induces four main paths to the target $u_{10}$, which are further split into 16 paths as the four paths pass through the second load balancer $u_5$. According to Definition 2, each of the 16 paths to the target requires a separate misery digraph, which can be expensive to implement. Optimizing misery digraph generation for paths that include load balancers involves the design of a load balancer system and splitting the connectivity digraph paths that involve load balancers.

First, the requirement for load balancers to function in a misery digraph is to balance the request traffic towards the least occupied successor machine, and *multicast* the response traffic back to all predecessor machines. For example, $u_5$ in Figure 3 must send the request to the least occupied machine in $\{u_6, \ldots, u_9\}$, and multicast the response to all machines in $\{u_1, \ldots, u_4\}$.

Second, since load balancers create overlapping paths towards the target server (as they must connect to multiple machines for fault tolerance), misery digraph redundancy is avoided by splitting paths at load balancers into subpaths. A load balancer is treated as a target for the incoming subpath, and as the predecessor to a machine that is treated as the entry point to an outgoing subpath. With this splitting technique, misery digraphs provide redundancies to frustrate attackers for each subpath without duplication from overlapping paths. The example depicted in Figure 5 demonstrates splitting of a path starting at the user $A$ and ending at the target database $u_{10}$, passing through the load balancers $u_0$ and $u_5$. The subpath $(A, u_0)$ is unchanged, and one misery digraph appears before $u_5$ and a second appears after $u_5$. The requests will only travel through one subtree of $u_5$ (only one shown in Figure 5) while responses are multicast back to all the vertices connected to $u_5$ in the layer above it.

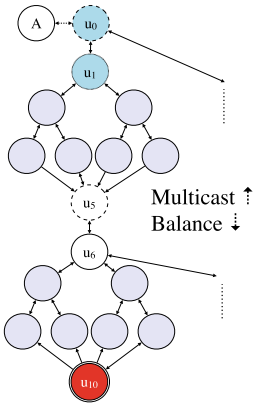



Fig. 5. Two misery digraphs are created for the HTTP path ($A$, $u_0$, $u_1$, $u_5$, $u_6$, $u_{10}$) for the connectivity digraph of Figure 3.

**Algorithm 1** Split a Connectivity Digraph $G = (V, A, \ell)$

```
for each s ∈ S do
  Set A_s ← ∅, ℓ_s ← ∅
  for each u ∈ V do
    for each v ≠ u ∈ V do
      if (u, v) ∈ A and s ∈ ℓ(u, v) then
        A_s ← A_s ∪ {(u, v)}
        ℓ_s ← ℓ_s ∪ {((u, v), {s})}
      end if
    end for
  end for
  Γ ← Γ ∪ {(V, A_s, ℓ_s)}
end for
return Γ
```

Load balancers are either part of the cloud provider or could be implemented as virtual machines. We do not assume that attackers cannot compromise load balancers. A compromised load balancer does not provide useful information to attackers.

## IV. Construction of Misery Digraphs

As defined in Section III, misery digraphs enlarge individual attack paths of a connectivity digraph that connects entry points to the target. A practical solution must combine misery digraphs into a new connectivity digraph (hereafter referred to as the final connectivity digraph) that can be deployed in the cloud, by first constructing an initial connectivity digraph given a cloud's firewall and network connection rules, and then generating misery digraphs that contain $k$-ary trees with $d + 1$ layers.

Assume that the connectivity digraph for an application only contains the necessary arcs for ensuring delivery of application requests and responses. At a high level, the construction of the final connectivity digraph for an initial connectivity-labelled digraph $G$ involves:

1) generating a set $\Gamma$ containing simple connectivity-labelled digraphs $G_s$ for each service $s$,
2) computing a set $P$ of subpaths of all paths connecting the entry vertex to a target vertex in $G$,
3) converting each $p \in P$ to a misery digraph, and
4) combining all the misery digraphs in a final connectivity digraph.

Next, we present the algorithms for executing the steps above and develop a cost analysis as a metric for establishing a baseline to evaluate the economic impact of using misery digraphs in a cloud.

### A. Constructing the Initial Graphs

The first step is to construct the initial connectivity-labelled digraph, $G$, from the application's architecture. We start by preparing a stack of machine IP addresses $M$ and a set of firewall inbound and outbound rules $R$ as $(m_1, m_2, s)$ indicating that the machine with IP address $m_1$ can access the machine with IP address $m_2$ on protocol $s$. This information is available from the cloud's console in platforms such as Amazon Web Services and Google Cloud Platform. Once the rules are gathered from the cloud's console, we construct a connectivity-labelled digraph $G$ by assigning a vertex to every $m \in M$ and adding a labelled arc $(u_i, v_j, L_{ij})$ where

$$L_{ij} = \bigcup_{(u_i, v_j, s) \in R} \{s\}.$$

There will likely be labels involving multiple services. To increase the efficiency of misery digraph construction, we split the cloud's connectivity digraph $G$ into simple connectivity digraphs in which every label $\ell(u, v)$ is the same single service $\{s\}$. Algorithm 1 splits a connectivity-labelled digraph $G$ into a set of simple connectivity-labelled digraphs, $\Gamma$. Each $G_s \in \Gamma$ is defined for a service $s \in S$, where $S$ is the set of all services appearing in $G = (V, A, \ell)$.

### B. Computing Paths to Target

The next step finds the paths for constructing misery digraphs. Recall that a misery digraph replaces a single path from an entry point to a target (Definition 2). Assuming the digraphs, $G_s \in \Gamma$, do not include unnecessary arcs and are not complete, finding all paths from each entry point to each target machine can be done efficiently using repeated calls to Dijkstra's shortest path algorithm.

Algorithm 2 examines the vertices in each $G_s$ to decide if a vertex is an entry point, which heads a subset of paths to the target. The function nextpath$(G_s, u, v)$ finds the next unique shortest path from $u$ to $v$ in $G_s$. In practice, entry point vertices can be stored in a list that includes all vertices that allow inbound access on an elastic IP address (accessible from outside the cloud) on the service $s$ for which the simple connectivity digraph $G_s$ is constructed.

Let $L_s \subseteq V_s$ be the set of load balancer vertices in $A_s$, $O_s \subseteq V_s$ denote the set of all entry point vertices in $A_s$, and $T_s \subseteq V_s$ denote the set of all target vertices in $A_s$.

The output of Algorithm 2 is the input to the final construction, which converts every (sub)path in $P$ to a misery digraph containing a canonical misery digraph. The union of the resulting misery digraphs will form the final connectivity digraph that can replace the original cloud connectivity digraph $G$.

**Algorithm 2** Find All Paths and Subpaths Between Entry Points, Load Balancers, and Targets in Each $G_s = (V_s, A_s, \ell_s) \in \Gamma$

```
Set P ← ∅
for each G_s ∈ Γ do
  for u ∈ O_s (entry point vertices) do
    for each v ∈ T_s (target vertices) do
      loop
        p ← nextpath(G_s,u,v)
        if p = ∅ then
          Break
        end if
        if p contains b_1, ..., b_j ∈ L_s then
          split p at b_1, ..., b_j into j + 1 subpaths (as
          described in III.C): p_1, p_2, ..., p_{j+1}
          P ← P ∪ {p_1, ..., p_{j+1}}
        else
          P ← P ∪ {p}
        end if
      end loop
    end for
  end for
end for
return P
```

### C. Constructing the Final Connectivity Digraph

The final construction takes each path $p \in P$ (an original path from an entry point to a target in the connectivity digraph $G$ or a subpath from the splitting in Algorithm 2) and replaces it with a misery digraph. Before the replacement, we expand all paths to be at least the minimum path length $d \geq \min_{p \in P} |p|$, where $|p|$ denotes the length of path $p$. We choose the fanout $k \geq 2$ of the canonical misery digraphs. Thus, this construction replaces every path in $P$ with a misery digraph containing a canonical misery digraph of $d + 1$ layers and fanout $k$.

The enlargement requires at most $\max\{0, d - |p|\}$ new vertices, and the canonical misery digraph requires another $(k^d - 1)/(k - 1) - d$ new vertices. The final result, after processing each $p \in P$ and taking the union of all these misery digraphs, is the misery digraph $G^*$ for the original connectivity digraph $G$.

### D. Additional Cost of Misery Digraphs

To evaluate the feasibility of misery digraphs as a defense, we need to understand the costs required by the defender relative to the increase in adversary cost. The main cost for the defense is the need for the decoy virtual machines, which must appear indistinguishable from the real hosts to intruders. We analyze the extra cost in terms of the increase in the hourly rate for the entire cloud-based network as a result of applying misery digraphs.

The cost of a cloud is modeled as a summation of the cost of all services used to operate the cloud. Let $s_i$ be a service in the cloud, including virtual machines, load balancers, storage instances, or database instances. For $M$ services, the total hourly cost of a cloud is:

$$\sum_{i=1}^{M} h(s_i) + n(s_i) + d(s_i),$$

where $h$ is the direct cost of the service (e.g., hourly rent of a virtual machine), $n$ is the networking cost of the service (e.g., hourly traffic usage of the service), and $d$ is the identity cost of the service (e.g., reserved IP addresses for facing the Internet).

With the current technology in major cloud providers (AWS and Google Cloud Platform), a misery digraph only increases the hourly direct cost of the cloud by

$$\sum_{i=M+1}^{M+N} h(s_i),$$

where $N$ is the number of added decoy vertices in the misery digraph. The network $n(s_i)$ and identity costs $d(s_i)$ are zero for all decoy vertices as they only use internal networking without Internet traffic charges or the cost of reserving public IP addresses.

Note that the choice of decoy virtual machines must be relative to the choice of machines in the original cloud. For example, when the original cloud runs virtual machines of medium capability (two cores and 4GB of memory), decoy servers in each path to target should have at least two cores and 4GB of memory. This is to avoid saturating the decoy virtual machines with a high number of requests received from more capable virtual machines in the network.

When using misery digraphs, the number of requests and responses in the network do increase (and must, as necessary for eliminating side-channel attacks). Thus, the original virtual machines and the decoy machines require extra networking capabilities. For example, for every request in an original path $p$ to the target, a user's request is represented once at the application layer. When using a misery digraph, the same request is multicast to all subsequent branches, and thus generates multiple responses (to hide the actual path). When implementing the multicasting service, each vertex will only wait for a single response and discard the rest. While this operation consumes extra bandwidth, since the networking is internal (within one data center), the extra cost is zero.

Finally, the total increased cost of a cloud with a misery digraph depends on the expansion parameters used for each canonical misery digraph. For each attack (sub)path $p$, the number of extra vertices is

$$e(p) = \max\{0, d - |p|\} + \frac{k^d - 1}{k - 1} - d \tag{1}$$

and the total extra cost is proportional to

$$\sum_{p \in P} e(p). \tag{2}$$

Note that misery digraphs consider only unique paths from entry points to the target machines. Thus, clearly, paths with overlapping vertices will not require additional decoy vertices.

## V. Security Analysis

Including a canonical misery digraph in every subpath guarantees that every path connecting an entry point vertex to a target machine vertex has length at least $d$ vertices and a misery fanout of at least $k^{d-1}$. The enlarged connectivity digraph will add complexity and time to an attack targeting a server that is required to be accessible only by a leaf of a $k$-ary tree in a misery digraph.

Recall that the attacker's goal is to compromise the database server by finding vulnerabilities in vertices along paths to a target. In a cloud that contains misery digraphs, assuming the attacker has no prior knowledge about the structure of the cloud, an attacker is likely to either attack the network by performing a depth-first attack or a breadth-first attack, because reaching the target server requires finding a path through which the intrusion can proceed. This section analyzes both attack strategies and estimates the delay incurred as a result of misery digraphs. Section V-D describes some attacks which are not mitigated by our approach.

### A. Resilience Against Reconnaissance Attempts

Reconnaissance attacks including DNS and IP scanning, operating system fingerprinting, examination of the cloud computing provider, and exploring the internal network architecture of a cloud are effective ways for attackers to launch informed attacks. When an intruder gains access to the cloud's entry point, launching an effective attack on the next layer of decoy machines includes two major steps.

*1) Collecting System Details:* The attacker collects technical systems-level visible details of the machines accessible from the entry points. The knowledge of the hosting cloud computing provider is necessary to predict the range of regional elastic IP addresses. Elastic IP addresses can identify decoy machines in misery digraphs but are not static. Periodic switching of attack paths (Section III-C) imposes a shuffling of these addresses, which is shown to be an effective general moving target strategy [15]. Further, the attacker is assumed to collect operating system signatures and configurations. The attacker sniffs the traffic when a node is compromised to view the flow of traffic, provided enough privileges are gained.

Reconnaissance information collected from details are necessary for the attacker to proceed. This information does not undermine the security provided by misery digraphs. This is because misery digraphs provide an architectural solution that does not rely on the specific functionality of machines. Further, misery digraphs allow for the probability that decoy machines are exploited by attackers. At each time period, when a switching occurs, any reconnaissance information or exploited machines on the switched machines are rendered obsolete.

*2) Reconstructing the Network Architecture View:* Once gained access to an entry point machine, the attacker attempts to construct an architectural view of the internal virtual cloud-based network. This search is itself a reconnaissance activity in which the attacker must use a graph search strategy to find the moving target.

Misery digraphs, as specified in Section III, are designed to provide *identical paths* towards any destination. This is to prevent attackers from pruning tree branches within a misery digraph's $k$-ary tree, thus, gaining a shortcut towards the target. The identical paths are provided using three critical design decisions:

1) The initial structure of misery digraphs provides equal numbers of vertices accessible from any vertex. No path in the digraph is distinctly identifiable in terms of its proximity to the target.
2) The network traffic maintained by misery digraphs also follows an identical distribution of requests. That is, each decoy forwards the requests to all successive machines without prioritizing or neglecting any machine. Similarly all responses are forwarded back up to the parent vertices.
3) Connections from any vertex $u_i$ to any other vertex $u_j$ cannot occurs unless $u_i$ is a direct parent of $u_j$. Accordingly, no vertex $u_i$ shall establish connections to a vertex $u_j$ if $u_i$ is more than one layer away. As this property is enforced by the rules set using the cloud computing provider, it ensures that attackers cannot construct shortcuts towards the target.

In Section V-B, two attack strategies are described for conducting an effective search against misery digraphs. Later in Section V-C, the probability of success for reaching a particular vertex is assessed, and finally in Section VI-A.1, a simulation evaluates the overall security of misery digraphs against these strategies.

### B. Attack Strategies

We first examine the available attack strategies, which provide the basis for a probabilistic analysis of attack success. These strategies are designed to search the structure of misery digraphs and outpace the moving target defense provided by the cloud. A depth-first attack (DFA), inspired by depth of stack routing [12], uses a depth-first search strategy to construct a single path towards a target starting with a vulnerable entry point. Next, the attacker is faced with a choice of machines to (i) test for vulnerabilities and (ii) craft an attack. To continue with a pure DFA, the attacker repeats the previous step by choosing one of the available IP addresses to attack. These repeated "compromise and choose" steps will continue until the attacker reaches a vertex that has an arc towards the target database. A key guarantee of clouds made with misery digraphs is that by examining the structure of the cloud, the attacker will not be able to make intelligent guesses about the next vertex to exploit.

A better approach is a breadth-first attack (BFA), inspired by breadth-first search algorithms for network routing (such as [2]), using which the attacker performs a breadth-first search to construct a path towards a target starting from a vulnerable entry point. Assuming the attacker has a set of IP addresses to invade next, the strategy in BFA will involve a per layer attack of all vertices in the graph until a leaf is found that has a direct arc to the target (which provides access to the target server). A BFA systematically explores the IP ranges available to the attacker. As misery digraphs contain $k$-ary tree structures, the available IP addresses will only enable a

layered attack. In a breadth-first attack, the attacker searches for an attack path by discovering the entire structure of the misery digraph.

### C. Swiching Probability

Consider a trivial connectivity digraph with one entry point and one target, replaced by a canonical misery digraph with $(k^d - 1)/(k - 1)$ vertices and $n = (k^d - 1)/(k - 1) - 1$ edges in the embedded $k$-ary tree. This probability analysis considers a breadth-first attack that randomly chooses a vertex to compromise at each level.

We analyze the probability that a breadth-first attack to reaches level $d$ (one step from the target) in a cloud with a randomized misery digraph. Let $D$ (delay) be the time required to compromise a vertex, and $r$ the period at which two random pairs of arcs at the same (random) level are interchanged (Definition 2). After each time period, we assume the vertices at the heads and tails of these arcs are reset to uncompromised states (which the attacker must compromise again).

For the time interval $D$ after an edge switch, the probability that a given edge $\{u, v\}$ is not switched is

$$\left(\frac{n-1}{n}\right)^{\lfloor D/r \rfloor}, \tag{3}$$

and the probability of not switching $m$ distinct given edges (required to maintain a path containing those $m$ edges for time $D$, so as to continue the attack from the last vertex in that path) is

$$\left(\frac{n-m}{n}\right)^{\lfloor D/r \rfloor}. \tag{4}$$

For example, with a delay $D = 0.5$, an edge switch period $r = 0.01$ (relative to some time unit), a misery digraph with $k = 2$ and $d = 5$ ($n = 30$), and a target compromising path of length $m = 4$, the probability of success would be 0.00078. If $D = 1.0$, that probability drops to $6 \cdot 10^{-7}$. The expected delay is roughly the reciprocal of this probability times $D$. Furthermore, note that even if the attacker is successful in reaching the target, access to the target is fleeting as it is only a matter of time before a path edge required by the attacker is switched and the machines on the two end points are reset (disrupting the attack and requiring repeated effort from the attacker).

The edge switch mechanism combined with the confounding architecture of the misery cloud significantly lowers the probability of reaching a vertex with direct access to the target. As a result, attacks are delayed depending on the ratio $D/r$, and by the misery digraph itself even without the randomization. In practice, the edge switch can be implemented in seconds, as fast as sending a request to the cloud provider and initiating a machine reset, thereby eliminating an entire path constructed by the attacker.

### D. Limitations

The goal of misery digraphs is to significantly delay an attack when the attacker's purpose is mainly to gain access to a database server in a cloud. Misery digraphs by themselves do not directly mitigate other types of attacks, but are generally complementary with defenses for other attacks.

*1) SQL Injection:* Misery digraphs do not target attacks that can only succeed using SQL injections into vulnerable applications. Such attacks do not rely on the structure of the network and cannot be defended against solely using an architectural solution like misery digraphs. However, misery digraphs can potentially couple with a parallel strategy that introduces diversity in various layers of the digraph. For example, if a vulnerable server in the entry point allows for a wide data query such as `SELECT * FROM t`, a diversified server in the next layer of the digraph can detect this. Thus, introducing diversity at each layer of the misery digraph can be a solution for attacks that succeed with simple requests.

*2) Denial of Service:* An attacker compromising a machine at any layer of the misery digraph may attempt denial of service by modifying the application, stopping the services, or similar approaches. Misery digraphs do not provide a solution for denial of service attacks, which have been heavily studied in the literature (Section VII). Because of the additional network traffic caused by multicasting between the layers, misery digraphs may even provide attackers with some additional opportunities for denial of service attacks.

*3) Compromised Cloud:* Misery digraphs depend on the integrity of the cloud. If the credentials for a cloud console are stolen, all security measures can be subverted. A misery digraph premise is that vertices will be compromised, but doing so takes significant time for *each* vertex along a path. Thus cloud users should avoid sharing credentials among machines.

## VI. Evaluation

This section aims to examine the effectiveness and practicality of misery digraphs. We first present an extensive simulation of the breadth-first attack against a changing misery digraph, showing that an estimated high delay in the attack. Then, we present a discussion on a prototype AWS misery digraph and the needed configuration. We implemented misery digraphs and the switching mechanism (Section III-C) using AWS Developer tools.[3] Finally, we demonstrate a concrete cost analysis based on running our prototype AWS misery digraph for a complete billing cycle.

The example network used in this section is based on a simplified network, having a web server and a database server. The network is then expanded using the construction algorithms in Section IV to create a misery digraph network.

### A. Measuring the Attack Success

Definition 2 requires that misery digraphs change over time. For example, in the misery digraph of Figure 6, as a function of time, the (underlying graph) edge $\{u_4, u_{10}\}$ is randomly chosen (using the cloud provider's tools) to switch with $\{u_6, u_{13}\}$, while the machines on both ends of each edge are reset and replaced with newly created machines on the fly. Given a misery digraph $G$, at any time, the attacker cannot guarantee

[3] https://aws.amazon.com/tools/

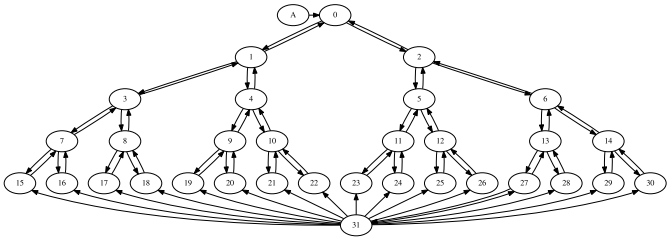



Fig. 6. A misery digraph with $k = 2$ and $d = 5$ used for simulating and measuring attack success.

that the observed structure of $G$ remains intact. Even if the attacker manages to predict the structure of $G$ by discovering the first few levels, as the edges are switched, the attacker's understanding of the misery digraph is soon obsolete. The edge switching mechanism modifies the path to the target, resulting in a loss of effort for an attack on the modified path.

*1) Attacker's Success in Outpacing the Defense:* To measure the expected delay caused by misery digraphs we implemented a discrete event simulation of an attack on a simple cloud architecture. The attack simulation's goal is to estimate the *attack success metric*, the number of hosts that must be compromised (or re-compromised) before an attack succeeds.

Simulation results are in two parts. The first part implements the attack strategies of Section V-B. The second part incorporates a branch pruning oracle, allowing the attacker to occasionally gain insider information indicating that the current path *does not lead to the target*.

*a) Attacking without pruning:* The simulation uses the misery digraph of Figure 6. A client attacker starts with vertex 0, which is assumed to be exploitable. The attacker builds a current understanding of the misery digraph $G^A$, which initially has $V = \{A\}$. The cloud is modeled as a server and has the initial misery digraph $G$, and also modifies $G$ every $r$ units of time. The attacker spends a constant $D$ units of time to compromise a vertex. When the attack starts, after spending time $D$, the attacker sends the path $(A, 0)$ to the cloud, indicating that 0 is compromised. To prove that the attacker has compromised the current 0 (before 0 is reset and replaced by a new machine), the cloud also requires the attacker to send 0's key. The cloud verifies the key and the path and responds with the vertices in the next layer of the misery digraph, that is, $\{1, 2\}$ along with their keys.

This interaction continues between the cloud and the attacker until time $r$ has elapsed and a change occurs in the misery digraph, which modifies a pair of edges and the machines on their ends. After the change to the digraph, if the attacker sends the cloud a path that was modified, the cloud detects this modification by either failing to verify a key or the path itself, responding to the attacker with an empty list. When the attacker receives an empty list, the attacker knows that $G^A$ is no longer consistent with $G$ and tries another vertex that was observed before. When all the observed vertices are tried without success, the attacker restarts at vertex 0.

The simulation was executed with parameters ($D = 0.1$, $r = 0.05$) and ($D = 0.2$, $r = 0.05$). Each experiment is repeated 1000 times, each time executing a complete cycle

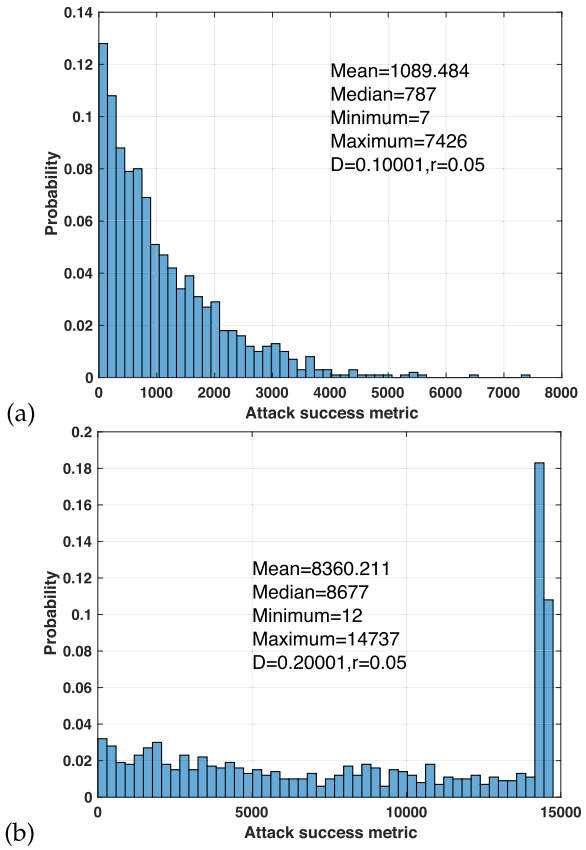



Fig. 7. Histogram of simulation results in which (a) the cloud is about twice faster than the attacker, and (b) the cloud is about four times faster than the attacker. The x-axis shows the attack success metric, the number of vertices the attacker tried until reaching the target. The y-axis shows the sample probability of the attack success metric.

of attack reaching the target. Although system configurations may prohibit actual repeated attacks, we did not implement this prohibition to test the strength of misery digraphs.

With $D = 0.1$, as the histograms of Figure 7 show, there is a 0.41 probability that the attack requires at least 1000 vertices before it reaches the target. In this case, during the 1000 iterations, the attack succeeded only four times with ten or fewer vertices. With $D = 0.2$, there is a 0.912 probability that the attack requires at least 1000 vertices to reach the target. The observed minimum number of attack steps increases from seven to twelve, with only a single time in 1000 iterations in which the attack succeeded with 20 or fewer steps.

The conclusion from the simulation results is that given a reasonably fast cloud modification procedure, the attack can take thousands of steps in a digraph where the actual shortest path to target consists of only five steps. Even a breadth first attack with a brute force strategy would only require 30 steps; however, with the misery digraph's structure and switching mechanism, these minimums are highly unlikely to occur.

*b) Attacking with pruning:* One might wonder if the attacker could use an oracle, which represents some leakage of the exploited machines, to decide if the current path will not lead to the actual target.

### B. Are Misery Digraphs Practical?

To test whether current technology permits creation of misery digraphs, we developed a tool that can connect to an Amazon Web Services cloud, download machine, connectivity, and firewall information, and create a connectivity digraph. The tool can transform an applications's connectivity digraph to a misery digraph. We have released the code under an open source license, available at `https://github.com/kussl/mdg`. This section empirically evaluates our misery digraph approach with respect to Amazon Web Services (AWS) using our tool.

AWS provides elastic virtual machines, IP addresses, virtual private clouds, customized routing rules, software-based firewalls, and load balancers, which all can help in building an application architecture that includes misery digraphs. AWS was used to create a cloud misery digraph with parameters $k = 2$ and $d = 3$, hosting a web application that queries a database and provides summary data.

The prototype misery digraph is created using basic AWS tools, mimicking a simplified web application architecture similar to the one in Figure 1. One EC2 instance (an AWS virtual machine), N1, is responsible for receiving requests from the Internet, and so is created with a subnet with an Internet gateway. In AWS, machines join security groups and the firewall rules can be configured for services provided by the machines in the group. Host N1 is in a single group that has the inbound HTTP rule allowing traffic from all IP addresses. As an externally-facing host, N1 does not include any credentials for the database. Instead, it will forward all database requests to the following layer containing two EC2 instances, N2 and N3. The EC2 instances N2 and N3 each include an Apache server and are only responsible for receiving and forwarding HTTP requests. These nodes are in a security group that allows inbound HTTP and SSH traffic only from N1. The hosts are configured so N2 forwards all requests to N4 and N5 while N3 forwards all requests to N6 and N7. Similar to the first internal layer (N2 and N3), N4 and N5 are in a security group that only allows inbound HTTP and SSH traffic from N2 while N6 and N7 are in a security group that only allows inbound HTTP and SSH traffic from N3. The reverse of all of the above traffic is allowed, but no traffic is allowed between N1 and N7. Only one of the four second-layer nodes (say N4 for this example) actually forwards its requests to N8, the target node. When N8 receives an HTTP request it processes it with a local MySQL database. Regardless of which second-layer node sent the request, N8 broadcasts the response on HTTP to *all* the second-layer nodes (N4–N7). N8 allows inbound HTTP and SSH traffic only with N4, and outbound HTTP traffic to N4–N7. The outbound rules for all machines are limited to only the necessary destinations.

*1) Relocating Decoys in Misery Digraphs:* As required by Definition 2 (in Section III-A), the incoming connection to the target machine (N8 in the implemented cloud) should be continuously interchanged between the final internal layer machines N4–N7. This randomness requirement can be implemented in AWS by dynamically modifying firewall security groups. That is, a single security group, which will have exactly one machine as a member at any time, can have access to N8. Each time period (whose length is determined by a time parameter set by the user and can be randomized), this security group is reset to contain one of the four machines N4–N7, chosen randomly. The other misery digraph arc switching described in Definition 2 was not implemented.

Another requirement is to reset the two machines at the heads and tails of a randomly chosen pair of arcs between two nodes within the underlying $k$-ary tree of a canonical misery digraph. To implement this we create a large set of configured machine images to choose from. To reset two machines, a cron task randomly chooses one of the machine images and launches two new instances using the AWS command line tool, run_instances. When the two instances are started, they are assigned to the security groups of the two old instances to be replaced (each pair of old and new instances will be matched). Finally, a call to delete_instances given the identifier for the two old instances terminates the old instances.

*2) Accommodating Existing Applications:* Existing applications can be deployed to use misery digraphs with minimal changes. Applications can continue to issue database requests to a mediating proxy machine that appears to the application server as the database server. The proxy machine will perform the broadcasting to the underlying misery digraph and forward the responses back to the application server. The proxy machine implements a simple proxy server, for example, Apache's `mod_proxy`. Using a proxy machine, drastic changes to the application are avoided. On the database server side, no changes will be necessary as the database server will continue to serve the requests coming from a leaf node (node in layer $d$) of a canonical misery digraph.

*3) A Concrete Cost Analysis:* The additional cost of misery digraphs is due to the use of decoy vertices, added to the network, which are realized as AWS EC2 instances. Misery digraphs incur no additional network charges since all the additional network traffic is within the internal network of the cloud. Consider network of five hosts with two entry points, two application servers, and a target. The connectivity digraph for the considered example cloud in Figure 8-a has two distinct paths and the digraph in Figure 8-b has four paths with overlapping vertices. Note that, misery digraphs are only created for distinct paths. That is, for both networks in Figure 8-a and 8-b an identical misery digraph will be created. This is because the overlapping paths $(1, 4, 5)$ and $(2, 4, 5)$ in Figure 8-b will share decoy vertices.

The original network of Figure 8 costs \$47.5 for a single billing cycle of 30 days (\$9.5 per machine[4]). We analyze the expected cost for replacing each path in the network of Figure 8, in Table II. In this table, we use a number of misery digraph parameters (first and second columns) to compute the extra vertices and cost (third and fourth columns) of each of the two misery digraphs needed for the example network. The extra vertices and cost are computed according to the formulas of Section IV-D. The fifth column shows the increase ratio with respect to the original cost of \$47.5. The last two columns

[4]https://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances/

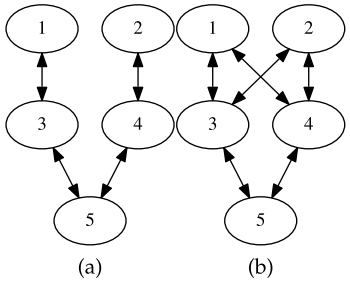



Fig. 8. Example networks with more than a single path each.

TABLE II

EXTRA VERTICES AND COST ARE COMPUTED USING EQUATION 2. THE INCREASE RATIO IS RELATIVE TO THE BASE COST OF $47.5 FOR THE ORIGINAL NETWORK BEFORE CONVERTING IT TO A NETWORK OF MISERY DIGRAPHS. THE VALUES OF $P$ ARE COMPUTED BASED ON EQUATION 4, REFERRING TO THE PROBABILITY THAT A SEQUENCE OF PATHS IS *not* SWITCHED DURING $r$ UNITS OF TIME. THE VALUES OF $Q$ ARE COMPUTED BASED ON THE SIMULATION OF SECTION VI-A.1, INDICATING THE PROBABILITY THAT THE ATTACK REQUIRES 500 OR MORE STEPS TO REACH THE TARGET WITHIN A NETWORK OF FIVE ORIGINAL MACHINES

| $k$ | $d$ | Extra vertices | Extra cost | Inc. Ratio | $P$ | $Q$ |
|---|---|---|---|---|---|---|
| 3 | 3 | 13 | $123.5 | 2.6 | 69.44% | 10% |
| 2 | 3 | 7 | $66.5 | 1.4 | 44% | 54% |
| 2 | 4 | 15 | $142.5 | 3 | 61.73% | 54% |
| 2 | 5 | 31 | $294.5 | 6.2 | 75% | 87% |

provide the probability that a sequence of $m$ edges are not switched during $r$ units of time, denoted $P$, and the probability that the attacker requires 500 or more compromises to reach the target, computed using the simulation of Section VI-A.1.

From the results of Table II, we present two conclusions. First, although misery digraphs can be costly, one does not need a large misery digraph for effectively confusing the attacker. As the simulation results demonstrate, with only seven extra vertices (per misery digraph), there is a 54% probability that the attacker needs to compromise or re-compromise 500 or more vertices before reaching the target. Second, the results of the first and the second rows of the table show that the choice of parameters can incur extra charges without improved results. In the first row where $k = d = 3$, there is an increased probability that a vertex is not switched, compared to when $k = 2$ and $d = 3$. However, as the misery digraph does not become long enough, the attacker can be more successful. A systematic formulation of cost versus the size of misery digraphs can assist system administrators to optimize their choices, which will be left for a future work.

## VII. PRIOR WORK

Misery digraphs establish a deceiving architecture in a virtual network that also actively uses a moving target strategy to distract powerful intrusions within the network. Prior work in the design of network overlays and in moving target defense has inspired and is closely related to the present work. However, no prior work has explicitly aimed to trap cloud intruders by delaying and complicating remote attacks. Misery digraphs do not require secret entities, do not perform traffic filtration, and address attacks beyond distributed denial of service. In this section we examine network overlays, a number of closely related moving target defense strategies and theoretical frameworks, and embark on approaches that have used decoys in other settings.

### A. Network Overlay and Deception

In a secure network overlay [25], the target node only communicates with verified sources. After verifying the source, a secret subset of nodes forward the verified traffic to the target. Secure Overlay Services (SOS) [18] and WebSOS [23] are classical approaches that use network overlays to defend a target against DDoS attacks. SOS is a deceiving architecture based on source filtration. WebSOS implements SOS replacing strong client authentication with graphic Turing tests. SOS and WebSOS target DDoS attacks and rely on secret nodes, while misery digraphs implement a layering approach without the need for filtration or secret nodes. Further, all these proposals explicitly target physical networks and do not use elastic replicas as in misery digraphs. Denial of Service Elusion (DoSE) [34] reuses the idea of overlay networks [27] in the cloud where virtual machines comprise overlay networks and a management layer repeatedly tries to distinguish legitimate from malicious clients. Misery digraphs, in contrast, do not use filtration or learning, and are neutral to the network traffic.

### B. Moving Targets

Moving target is an effective technique that incorporates diversity and shuffling to achieve higher security [11], [14]. When a target machine is under attack, Migrating OVErlay (MOVE) [27] relocates the target machine's service to an unaffected machine and, as opposed to SOS [18], does not require client filtration. MOVE relies on hidden servers and also uses an overlay network to distinguish unknown traffic from legitimate traffic. Venkatesan *et al.* [30] address intercepting exfiltrated data using a moving target defense (backed by a probabilistic analysis) by dynamically replacing intrusion detection sensors. Their threat model assumes the attacker can explore the network topology and is aware of the moving target defense. MOTAG [16], on the other hand, uses moving secret proxies to distinguish attackers from legitimate clients. MOTAG's core idea is to provide a single secret IP address to a legitimate client, at any given time. The target servers only allow incoming traffic from designated proxies that are meant to be reachable by legitimate clients. Comparing to MOTAG, misery digraphs do not require secret proxies and mainly rely on a trusted cloud console that controls the policies and structure of the cloud's internal network. Also, it is shown that proxies can be subject to proxy harvesting attack, which require continuous remapping to disrupt the attacks [29]. Badishi *et al.* [4] proposed random port hopping to keep a DDoS attacker in the dark while using packet filtration to recognize legitimate traffic. Similarly, redundant data routing paths [19] can distract attackers from their favorable targets. Rather than relocating the target machine as in the defenses

against DDoS attacks, misery digraphs change the path to the target machine as a continuous function of time.

Apart from physical distributed systems, moving target defense promises a viable strategy for securing elastic clouds. The work by Brzeczko *et al.* [5] and Jia *et al.* [17] are closest to our work in using cloud technologies and moving the target away from attacks. However, misery digraphs target intrusions within virtual networks as opposed to those targeting the surface. Brzeczko *et al.* [5] demonstrate an analytical method that uses decoys in an elastic cloud computing platform to attract attacker traffic. The proposed system will then learn and redirect the malicious traffic from the production machines by using the data collected from decoy machines. In 2014, Jia *et al.* [17] presented the architecture of a system that uses a moving target defense for Amazon Web Services. To rescue targets in a virtual network from DDoS attacks, a defense system creates replicas on the fly and assigns network traffic to new replicas. A key assumption of the approach is hiding the newly created replicas from the public and disclosing their addresses to a select list of clients. Misery digraphs do not directly respond to attacks and avoid problems such as Economic Denial of Sustainability [33], which would cause unnecessary charges on the cloud.

### C. Theoretical Frameworks

Some recent work provides interesting theoretical frameworks for various moving target defense settings. For example, Wright *et al.* [35] evaluated moving defense strategies using a game-theoretic simulation, deriving insights for scenarios where a moving target is useful for combating DDoS attacks. Miehling *et al.* [22] present Bayesian attack graphs and model the defender's action as a partially observable Markov decision process in which some of the attacker's actions are not clear. The proposed Bayesian model limits the capabilities of an attacker by assuming a sequence of completely random attack steps. Our work assumes a more accurate representation of an intelligent attacker that will take many informed attack steps. Zhuang *et al.* [36], [37] also presented an inspiring theory of moving target defense in which they proposed a general system and an initial underlying theory for moving target defense. While having fundamentally different goals, this work on theoretical aspects of moving target defense is related to the present efforts. Maleki *et al.* [21] described a general theory on assessing the effectiveness of a moving target strategy, which is useful in conjunction with the probabilistic analysis of Section V-C.

### D. Other Uses of Decoys

The core aspect of misery digraphs is the use of moving decoys to create deception by hiding the true path towards a target database, somewhat different from the deception in some previous work where the use of decoys has been heavily discussed. For example, [31] proposes the use of multiple decoys, such as decoy HTML documents to distract attackers. Similarly, Voris *et al.* [32] demonstrate how decoy files can distract attackers from the target. Interesting work by Araujo *et al.* [3] proposes honey patches that confuse attackers about whether a software exploit has succeeded. This work might be especially useful when deployed in conjunction with misery digraphs. In a theoretical analysis, Pawlick and Zhu [24] demonstrate, through cheap-talk games, that honeypots could be used to create deception for attackers. An earlier game theoretic investigation of honeypots is described in [6]. A recent work by Luo *et al.* [20]proposes the use of dynamic path identifiers for network routing that dynamically change to escape DDoS attacks. Finally, Heydari *et al.* [13]demonstrated the use of moving target defense in web servers, acting as mobile nodes, to combat Internet censorship.

## VIII. Conclusion

Misery digraphs use the cloud's elastic and cost-effective services to deceive and frustrate attackers. A graph theoretic model that includes multiple redundant paths towards a cloud target was proposed and implemented in AWS. The idea of using redundancy to distract attackers does not intend to completely eliminate an attack, but to force enough delay on an aggressive attack to give system administrators time to intercede in the attack. Thus the delay and confusion and obscurity mechanisms provide the architectural support for a cloud to defend itself until rescue arrives.

An overall target defense strategy would require an effective intrusion detection mechanism that can collaborate with the misery digraphs and a mechanism to prevent an intrusion from reaching the target. Future extensions of this work might enable the misery digraphs themselves to act as detectors of intrusion, e.g., using the redundant paths as sensors to warn an outside monitor of possible attacks. For instance, *malicious* SSH connections to the redundant machines could trigger such an alarm. Detecting intrusions using misery digraphs will be addressed in future work.

## References


[1] (Aug. 2014). *An In-Depth Analysis of SSH Attacks on Amazon EC2*. Accessed: Feb. 1, 2017. [Online]. Available: https://blog.smarthoneypot.com/in-depth-analysis-of-ssh-attacks-on-amazon-ec2/

[2] B. Abali and C. Aykanat, "Routing algorithms for IBM SP1," in *Proc. 1st Int. Workshop Parallel Comput. Routing Commun. (PCRCW)*, 1994, pp. 161–175.

[3] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2014, pp. 942–953.

[4] G. Badishi, A. Herzberg, and I. Keidar, "Keeping denial-of-service attackers in the dark," *IEEE Trans. Depend. Sec. Comput.*, vol. 4, no. 3, pp. 191–204, Jul. 2007.

[5] A. Brzeczko, A. S. Uluagac, R. Beyah, and J. Copeland, "Active deception model for securing cloud infrastructure," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2014, pp. 535–540.

[6] T. E. Carroll and D. Grosu, "A game theoretic investigation of deception in network security," in *Proc. 18th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Washington, DC, USA, Aug. 2009, pp. 1–6.

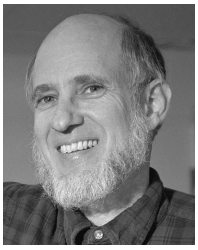
[7] L. Cheng, F. Liu, and D. D. Yao, "Enterprise data breach: Causes, challenges, prevention, and future directions," *Data Mining Knowl. Discovery*, vol. 7, no. 5, p. e1211, 2017.

[8] A. Chowdhary, S. Pisharody, and D. Huang, "SDN based scalable MTD solution in cloud network," in *Proc. ACM Workshop Moving Target Defense (MTD)*, New York, NY, USA, 2016, pp. 27–36.

[9] The Apache Struts Project Management Committee. (2017). *Apache Struts Statement on Equifax Security Breach*. [Online]. Available: https://blogs.apache.org/foundation/entry/apache-struts-statement-on-equifax

[10] *National Vulnerability Databas*, Document CVE-2017-9805 Detail, 2017. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-9805
[11] D. Evans, A. Nguyen-Tuong, and J. Knight, *Effectiveness of Moving Target Defenses*. New York, NY, USA: Springer, 2011, pp. 29–48.
[12] A. Gupta, A. Kumar, and M. Thorup, "Tree based MPLS routing," in *Proc. 15th Annu. ACM Symp. Parallel Algorithms Archit. (SPAA)*, New York, NY, USA, 2003, pp. 193–199.
[13] V. Heydari, S.-I. Kim, and S.-M. Yoo, "Scalable anti-censorship framework using moving target defense for Web servers," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 5, pp. 1113–1124, May 2017.
[14] J. B. Hong and D. S. Kim, "Assessing the effectiveness of moving target defenses using security models," *IEEE Trans. Depend. Sec. Comput.*, vol. 13, no. 2, pp. 163–177, Mar. 2016.
[15] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "An effective address mutation approach for disrupting reconnaissance attacks," *IEEE Trans. Inf. Forensics Security*, vol. 10, no. 12, pp. 2562–2577, Dec. 2015.
[16] Q. Jia, K. Sun, and A. Stavrou, "MOTAG: Moving target defense against Internet denial of service attacks," in *Proc. 22nd Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2013, pp. 1–9.
[17] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and W. Powell, "Catch me if you can: A cloud-enabled DDoS defense," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, Jun. 2014, pp. 264–275.
[18] A. D. Keromytis, V. Misra, and D. Rubenstein, "SOS: An architecture for mitigating DDoS attacks," *IEEE J. Sel. Areas Commun.*, vol. 22, no. 1, pp. 176–188, Jan. 2004.
[19] P. P. C. Lee, V. Misra, and D. Rubenstein, "Distributed algorithms for secure multipath routing in attack-resistant networks," *IEEE/ACM Trans. Netw.*, vol. 15, no. 6, pp. 1490–1501, Dec. 2007.
[20] H. Luo, Z. Chen, J. Li, and A. V. Vasilakos, "Preventing distributed denial-of-service flooding attacks with dynamic path identifiers," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 8, pp. 1801–1815, Aug. 2017.
[21] H. Maleki, S. Valizadeh, W. Koch, A. Bestavros, and M. van Dijk, "Markov modeling of moving target defense games," in *Proc. ACM Workshop Moving Target Defense (MTD)*, New York, NY, USA, 2016, pp. 81–92.
[22] E. Miehling, M. Rasouli, and D. Teneketzis, "Optimal defense policies for partially observable spreading processes on Bayesian attack graphs," in *Proc. 2nd ACM Workshop Moving Target Defense (MTD)*, New York, NY, USA, 2015, pp. 67–76.
[23] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein, "Using graphic turing tests to counter automated DDoS attacks against Web servers," in *Proc. 10th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2003, pp. 8–19.
[24] J. Pawlick and Q. Zhu, "Deception by design: Evidence-based signaling games for network defense," *CoRR*, abs/1503.05458, 2015.
[25] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 59–64, Jan. 2003.
[26] A. Stasinopoulos, C. Ntantogian, and C. Xenakis, "Commix: Detecting and exploiting command injection flaws," Dept. Digit. Syst., Univ. Piraeus, Piraeus, Greece, White Paper, Nov. 2015.
[27] A. Stavrou, A. D. Keromytis, J. Nieh, V. Misra, and D. Rubenstein, "MOVE: An end-to-end solution to network denial of service," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, 2005, pp. 81–96.
[28] Z. Su and G. Wassermann, "The essence of command injection attacks in Web applications," in *Proc. Conf. Rec. 33rd ACM SIGPLAN-SIGACT Symp. Principles Programm. Lang. (POPL)*, New York, NY, USA, 2006, pp. 372–382.
[29] S. Venkatesan, M. Albanese, K. Amin, S. Jajodia, and M. Wright, "A moving target defense approach to mitigate DDoS attacks against proxy-based architectures," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, Oct. 2016, pp. 198–206.
[30] S. Venkatesan, M. Albanese, G. Cybenko, and S. Jajodia, "A moving target defense approach to disrupting stealthy botnets," in *Proc. ACM Workshop Moving Target Defense (MTD)*, New York, NY, USA, 2016, pp. 37–46.
[31] N. Virvilis, B. Vanautgaerden, and O. S. Serrano, "Changing the game: The art of deceiving sophisticated attackers," in *Proc. 6th Int. Conf. Cyber Conflict (CyCon)*, Jun. 2014, pp. 87–97.
[32] J. Voris, J. Jermyn, N. Boggs, and S. Stolfo, "Fox in the trap: Thwarting masqueraders via automated decoy document deployment," in *Proc. 3th Eur. Workshop Syst. Secur. (EuroSec)*, New York, NY, USA, 2015, pp. 3:1–3:7.
[33] H. Wang, Z. Xi, F. Li, and S. Chen, "Abusing public third-party services for EDoS attacks," in *Proc. 10th USENIX Conf. Offensive Technol. (WOOT)*, Berkeley, CA, USA, 2016, pp. 155–167.
[34] P. Wood, C. Gutierrez, and S. Bagchi, "Denial of service elusion (DoSE): Keeping clients connected for less," in *Proc. IEEE 34th Symp. Reliable Distrib. Syst. (SRDS)*, Sep. 2015, pp. 94–103.
[35] M. Wright, S. Venkatesan, M. Albanese, and M. P. Wellman, "Moving target defense against DDoS attacks: An empirical game-theoretic analysis," in *Proc. ACM Workshop Moving Target Defense (MTD)*, New York, NY, USA, 2016, pp. 93–104.
[36] R. Zhuang, A. G. Bardas, S. A. DeLoach, and X. Ou, "A theory of cyber attacks: A step towards analyzing MTD systems," in *Proc. 2nd ACM Workshop Moving Target Defense (MTD)*, New York, NY, USA, 2015, pp. 11–20.
[37] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proc. 1st ACM Workshop Moving Target Defense (MTD)*, New York, NY, USA, 2014, pp. 31–40.

**Hussain M. J. Almohri** received the B.S. degree in computer science from Kuwait University and the Ph.D. degree in computer science from Virginia Tech. He is currently an Assistant Professor of computer science with Kuwait University and a Visiting Scholar with the University of Virginia. He has cofounded a mobile payment startup and has advised a number of software startups in the Gulf region. His research focuses on systems and network security. He has served as a reviewer for several IEEE and IET journals and the *Kuwait Journal of Science*.



**Layne T. Watson** (M'84–SM'89–F'93–LF'16) received the B.A. degree *(magna cum laude)* in psychology and mathematics from the University of Evansville, Evansville, IN, USA, in 1969, and the Ph.D. degree in mathematics from the University of Michigan, Ann Arbor, MI, USA, in 1974.

He was with USNAD Crane, Sandia National Laboratories, and General Motors Research Laboratories and has served on the faculties of the University of Michigan, Michigan State University, and the University of Notre Dame. He is currently a Professor of computer science, mathematics, and aerospace and ocean engineering with Virginia Tech. He has authored or co-authored over 300 refereed journal articles and 200 refereed conference papers. His current research interests include fluid dynamics, solid mechanics, numerical analysis, optimization, parallel computation, mathematical software, image processing, and bioinformatics. He is a fellow of the National Institute of Aerospace and the International Society of Intelligent Biological Medicine. He serves as a Senior Editor for *Applied Mathematics and Computation*, and an Associate Editor for *Computational Optimization and Applications*, *Evolutionary Optimization*, *Engineering Computations*, and the *International Journal of High Performance Computing Applications*.

**David Evans** received the S.B., S.M., and Ph.D. degrees in computer science from MIT. He has been a Faculty Member with the University of Virginia since 1999. He is currently a Professor of computer science with the University of Virginia and a Leader of the Security Research Group. He is currently the author of an open computer science textbook and a children's book on combinatorics and computability. He was the Program Co-Chair for the 31st (2009) and 32nd (2010) IEEE Symposia on Security and Privacy (where he initiated the SoK papers). He is the Program Co-Chair for the ACM Conference on Computer and Communications Security 2017.