Software Deception Steering through Version Emulation

Frederico Araujo
IBM Research
frederico.araujo@ibm.com

Sailik Sengupta
Arizona State University
sailiks@asu.edu

Jiyong Jang
IBM Research
jjang@us.ibm.com

Adam Doupé
Arizona State University
doupe@asu.edu

Kevin W. Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

Subbarao Kambhampati
Arizona State University
rao@asu.edu

Abstract

Determined cyber adversaries often strategize their attacks by carefully selecting high-value target machines that host insecure (e.g., unpatched) legacy software. In this paper, we propose a moving-target approach to thwart and countersurveil such adversaries, wherein live (non-decoy) enterprise software services are automatically modified to deceptively emulate vulnerable legacy versions that entice attackers. A game-theoretic framework chooses which emulated software stacks, versions, configurations, and vulnerabilities yield the best defensive payoffs and most useful threat data given a specific attack model. The results show that effective movement strategies can be computed to account for pragmatic aspects of deception, such as the utility of various intelligence-gathering actions, impact of vulnerabilities, performance costs of patch deployment, complexity of exploits, and attacker profile.

1. Introduction

Software vulnerabilities are among the top targets of cyber criminals, corresponding to 25% of all exploitable attack vectors [1]. The problem is exacerbated by the ready availability of exploits and detailed bug reports, which enable attackers to automate low-risk reconnaissance steps and probe victim networks for these software flaws. When a vulnerable target is identified, the attacker launches an exploit to the unpatched server, such as one that hijacks the victim software's control-flow, causing it to perform malicious actions on behalf of the attacker. The exploit payload thereby obtains access to the environment and deploys other malicious tools.

To anticipate and foil these directed cyber attacks, deceptive *honey-patching* [2] has been proposed as a language-based methodology to patch software security vulnerabilities in such a way that future attempted exploits of the patched vulnerability appear successful to attackers even when they are not. This masks patching lapses, impeding attackers from discerning which systems are genuinely vulnerable and which are actually patched systems masquerading as unpatched systems. Detected attacks are surreptitiously redirected to isolated, unpatched decoy environments with the full interactive power of the targeted victim server. The decoy environments disinform adversaries with honey-data and aggressively monitor adversarial behavior.

While these capabilities offer potentially promising defense layers against determined adversaries skilled at evading traditional honeypots, the question of which vulnerabilities to deceptively emulate or how to automatically evolve the deceptive attack surface with evolving adversarial TTPs has remained relatively unstudied. For example, Chameleon [3], Honeyd scripts [4], and Cloxy [5] deployments all presently rely upon human selection of vulnerabilities and versions, which is sub-optimal for waging long-term deceptive campaigns, because it can result in deceptions becoming predictable and stale, potentially affording attackers the opportunity and time to fingerprint and circumvent deceptive applications. As new vulnerabilities emerge, attack activity change [6, 7, 8, 9], potentially rendering old deceptions less enticing to cyber criminals.

To overcome this disadvantage, this paper proposes *software deception steering* as a new moving target defense technique for counterreconnaissance and attack intelligence gathering, which leverages application-level, deceptive attack responses through honey-patching to continuously adapt the *deception surface* of the target application. Toward this end, we designed and implemented QUICKSAND, an adaptive software version emulation architecture, in which the set of fake vulnerabilities is dynamically re-selected to increase the likelihood of deceiving and entrapping attackers. Based on vulnerability context (e.g., vulnerability risk scores, attack history), QUICKSAND chooses to emulate a particular software version with a particular set of (fake) vulnerabilities. This *moving* deception surface undermines the attacker's ability to identify and detect deceptions, and increases the likelihood of gathering *high-quality* threat data reflective of advanced attacks by skilled adversaries (rather than merely well-known attacks by unskilled adversaries, for which threat data is less useful).

Our work includes the following contributions:

- We propose a deception-based moving target architecture to dynamically honey-patch software, rendering it less predictable and more robust against attackers' anti-deception efforts.
- We model the software emulation process as a Bayesian Stackelberg Game [10, 11] to compute

effective movement strategies that account for pragmatic aspects of deception, including the utility of intelligence-gathering actions, impact of vulnerabilities, cost of patch deployment, complexity of exploits, and attacker model.

- We propose, design, and implement an effective version-control strategy to facilitate patch re-selection and automatically resolve source-level conflicts between patches.

2. Overview

First, we outline or new moving deception maneuver, followed by primary challenges and corresponding design decisions for software deception steering. Finally, we summarize background literature and our threat model.

2.1. Software Deception Steering

We define *deception steering* as the use of cyberdeception for altering the *apparent* attack surface of software systems towards configurations that yield better defender payoffs. Specifically, leveraging vulnerability metadata and intrusion alerts collected at the network perimeter, QUICKSAND dynamically adapts the target application to emulate a particular software version, with a particular set of honey-patched vulnerabilities (and all other known vulnerabilities regular-patched), a particular set of modules enabled, and a particular guest OS version in decoys.

The scope of adaptation can go beyond the application and host boundaries; for instance, perimeter defenses (if any) can also be reconfigured to intentionally allow previously filtered attacks to reach the honey-patch. This reconfiguration need not happen live; it can be re-selected during nightly reboots, for example. The selections are based on which configuration is likely to gather the most useful threat data given the history of past attacks.

2.2. Design Principles

Software deception steering requires a patch management framework that facilitates software version composition and minimizes source code-level conflicts between patches. QUICKSAND defines honey-patches as modifications to their corresponding regular, vendor-supplied patches. For instance, Figure 1 exemplifies a vulnerability causing the GNU Bash shell to improperly parse function definitions in the values of environment variables [12]. Prior to the patch, the vulnerable shell interpreter allowed remote attackers to execute arbitrary code or cause a denial of service on the victim's machine. The patch, named CVE-2014-6277 in this example, fixes the vulnerability by extending the check for what constitutes a legal function identifier to include some extra sanity checks (Lines 2–3 in the patch code, depicted in diff style). The honey-patch CVE-2014-6277-hp modifies the original patch to fork attacks onto decoy environments while impersonating the unpatched code (Lines 8–9 in the honey-patch code) to deceive adversaries. Encoding honey-patches in this manner naturally models the dependency among honey-patches, their corresponding patches, and unpatched source code. It also makes patch/honey-patch pairs conflict-free by construction, greatly simplifying the task of composing new versions of the target application.

Patch dependencies (denoted by dashed arrows between patches) are calculated based on how patches affect source code rather than by the order in which they are introduced into the code base. This removes the temporal constraint among patches and enables the selection of patch sets based on their *semantic* dependencies. This patch dependency model is implemented in the Darcs version control system [13], which our software version-emulation architecture leverages to select consistent, conflict-free application versions for deployment.

2.3. Background

Honey-patching. Prior work has observed that many vendor-released software security patches can be honeyed by replacing their attack-rejection responses with code that instead maintains and forks the attacker's connection to a confined, unpatched decoy [2, 14]. Such honey-patching retains the most complex part of the vendor patch (the security check) and replaces the remediation code with some boilerplate forking code [15], making it easy to implement upon release of new security patches.

This embedded deception offers some important advantages over conventional honeypots. Most significantly, it observes attacks against the defender's genuine assets, not merely those directed at fake assets that offer no legitimate services. It can therefore capture data from sophisticated attackers who monitor network traffic to identify service-providing assets before launching attacks, who customize their attacks to the particular activities of targeted victims (differentiating genuine servers from dedicated honeypots), and who may have already successfully infiltrated the victim's network before their attacks become detected.

Threat Model. Attackers in our model submit malicious inputs intended to probe and exploit vulnerabilities on victim networked services. We assume most attackers rely upon a mix of vulnerabilities, only some of which are known to defenders. For example, a skilled attacker might first try to exploit known vulnerabilities, only escalating to more potent, defender-unknown vulnerabilities (e.g., 0-days) once he becomes confident that his activities are not being observed. Attack payloads might be completely unique and therefore unknown to defenders. Such payloads might elude network-level monitors, and are therefore best detected at the software level at the point of exploitation. We also assume that attackers might use one payload for reconnaissance but reserve another for the final attack. Misleading the attacker into launching the final attack is useful for defenders to discover the final attack payload, which can divulge attacker TTPs and goals not discernible from the reconnaissance payload alone.

While our general approach is potentially applicable to arbitrary networked software, in this work we focus on

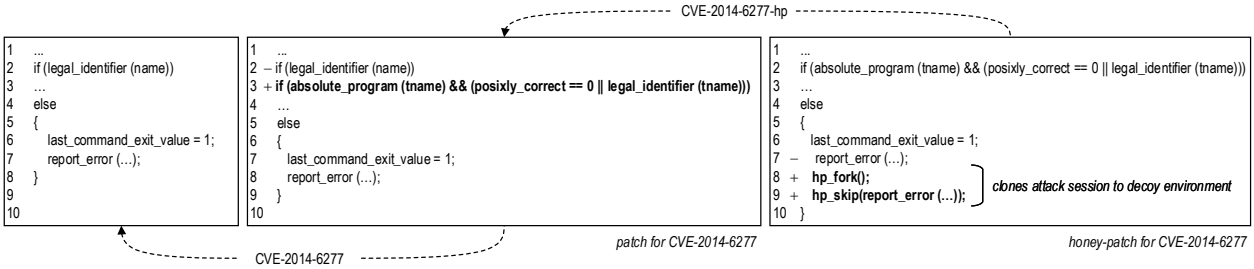

Figure 1: Patch and honey-patch for CVE-2014-6277 (abbreviated), and dependencies between them denoted by dashed arrows

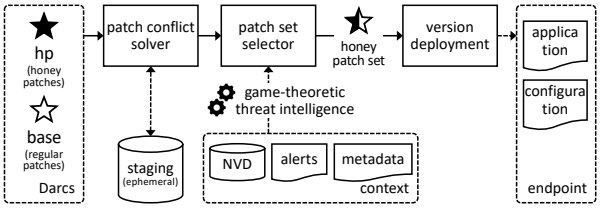

Figure 2: An overview of QUICKSAND.

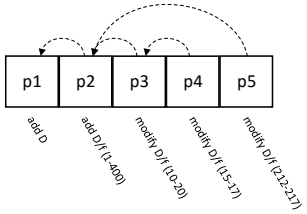

Figure 3: A repository state showing patch dependencies.

protecting services possessing strictly user-level privileges, and that must therefore leverage software bugs and kernel-supplied services to perform malicious actions, such as corrupting the file system or accessing other users' memory to access confidential data. QUICKSAND therefore instruments user-level applications with deceptive defensive code without modifying the OS or VM.

3. System Design

Figure 2 depicts QUICKSAND's architecture. The *patch conflict solver* generates conflict-free candidate patch sets for version emulation. An *analysis and correlation* component ingests and maintains vulnerability metadata from the National Vulnerability Database (NVD), parses intrusion alerts, and correlates them with intrusion signature metadata. The *patch set selector* module leverages a game-theoretic engine to select which version of the software should be deployed based on the aggregated data. The *version deployment* module then uses this information to synthesize and deploy a new version of the application, including the specification of the target modules and environment. This process executes repeatedly, and its trigger threshold can be fixed, random, or dynamically adjusted (e.g., based on evidence and severity of intrusion alerts collected at the network perimeter).

Patch Theory. Darcs is a *change-based* version control system. In contrast to conventional history-based version control systems (e.g., Subversion, Git, CVS), which represent repository states as file trees, the state of a Darcs repository is defined by the set of patches it contains. This facilitates a *cherry-picking* operation—one that is not constrained by temporal dependencies among patches—that adds flexibility to our patch set selection model. Cherry-picking can be defined in terms of Darcs' underlying patch theory [16, 17], summarized as follows:

Definitions. The state of a repository is called a *context*. We write ${}^{o}A^{a}$ to denote that a repository moves from context o to context a via patch A. Patches are usually stored sequentially, and for any consecutive pair of patches, the final state of the first patch must be identical to the initial state of the second patch. A sequence of patches is written in left to right order, such as ${}^{o}A^{a}B^{b}C^{c}$ (or simply ABC if we omit contexts). Parallel patches share a common initial context and diverge to two different states ($A \vee B$).

Inversion. Every Darcs patch is invertible, affording the application of patches in either forwards or backwards directions to reach a particular context: $(AB)^{-1} = B^{-1}A^{-1}$. In particular, AA^{-1} has no effect, and $(A^{-1})^{-1} = A$. Anti-parallel patches have different initial states yielding the same context ($A^{-1} \wedge B^{-1}$).

Commutation. The commutation of patches A and B is represented by $AB \leftrightarrow B'A'$, where A' and B' are intended to perform the same change as A and B. Intermediate states may differ however: ${}^{o}A^{a}B^{b} \leftrightarrow {}^{o}B'^{x}A'^{b}$. A *merge* operation is defined as a pairwise commutation, taking two *conflict-free* parallel patches and converting them into a pair of sequential patches: $A \vee B \Rightarrow AB' \leftrightarrow BA'$.

Cherry picking. Patch *cherry picking* refers to the ability to pull patches from a repository regardless of the order in which they were originally pushed into the repository. To illustrate, consider the repository state depicted in Figure 3. The repository consists of patches `p1`–`p5`, and the changes made by each patch are summarized underneath each patch. The dependencies between patches (denoted by dashed arrows) are computed by Darcs. Figure 4a illustrates cherry picking for this particular example. Pulling patches `p1`, `p2`, and `p5` from the *source* onto the *destination*

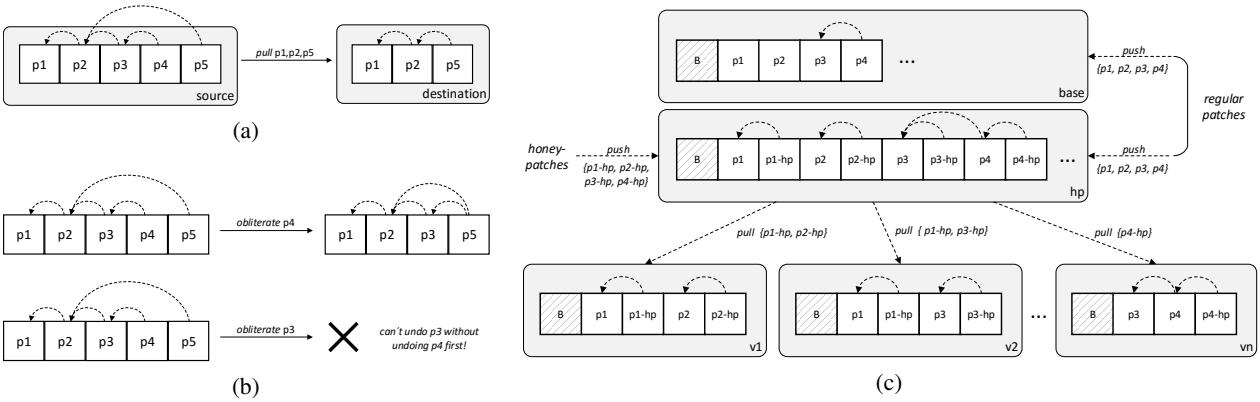

Figure 4: Operations illustrating (a) change-based patch cherry picking, (b) patch obliteration and consistency, and (c) patch management: patch set `B` denotes the set of patches making up the base source code of the software; patch dependencies pointing to it have been omitted.

repository automatically adjusts the selected patches to fit the new context (without `p3` and `p4`). Darcs performs such adjustments using its patch manipulation algebra to allow users to reason about patches as sets, despite patches being stored as sequences internally.

Patch obliteration and consistency. Another advantage of patch commutativity is that patches can be *obliterated* (undone) without rolling back patches that historically succeed them. In the example above, patch `p4` can be removed from the repository without undoing `p5`, as illustrated in Figure 4b. To accomplish this, Darcs rearranges the sequence of patches by commuting `p4` with `p5`, and then removes `p4`. However, Darcs does not allow `p3` to be removed without first undoing `p4`; allowing this operation would constitute a patch dependency violation and render the state of the repository inconsistent.

Patch Management. Figure 4c illustrates our patch management strategy. Regular patches are *pushed* (stored) into Darcs repositories `base` and `hp`, and honey-patches are stored into repository `hp` only. We call `B` the set of patches (i.e., all code changes) that constitute the base version of the software (e.g., the initial commit, a specific tagged version of the application containing all patches up to the tag). Candidate versions selected by the path selection module are stored as tags (e.g., `v1`–`vn`) by *pulling* specific patch sets from `hp`, which allows them to be easily retrieved for version deployment.

This patch management strategy leverages the underlying Darcs infrastructure, which automatically computes the transitive dependency relations for any given patch selection. For example, when pulling honey-patch `p4-hp`, Darcs correctly pulls patch set `B` and patches `p3`, `p4`, and `p4-hp`. This has the advantage of enabling a much simpler patch set generation algorithm (see §4).

Alert Analysis and Correlation. The alert analysis and correlation workflow pre-processes vulnerability and environmental data to generate contextual information for patch set selection. First, intrusion alerts are parsed, and each alert class is annotated with descriptive statistics and target information. In the second step, the correlation module parses the intrusion detection system's signature map to extract the signature information for each alert object and cross-references it with the corresponding CVE identification derived from the reference field specified in the alert metadata. This step additionally filters intrusion alerts whose signatures target vulnerabilities that have not been identified as CVEs. The last step consults vFeed [18] to look up common vulnerability and exploit databases (e.g., CVSS, CWE, exploit-db) to aggregate threat intelligence metadata (e.g., vulnerability risk scores, exploit availability) and alert objects, which are used by the game-theoretic decision engine during the version selection process. Listing 1 shows an alert object containing threat metadata for CVE-2014-6277.

Listing 1: Alert object containing threat metadata

```
1 { cveID: CVE-2014-6277,
2   targets: {('192.168.134.150', 80), ('192.168.134.139', 8080), ...},
3   cvss: { 'vector': 'CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/...'},
4   cwe: { id: 'cwe-78', term: 'OS_Command_Injection' },
5   cpe: { 'cpe:/a:gnu:bash:2.02.1', 'cpe:/a:gnu:bash:2.01.1', ...},
6   published: 2014-09-24T14:48:04.477-04:00,
7   public_exploit: 'yes',
8   count: 115 }
```

4. Software Deception Steering

To enable a truly dynamic system that makes it difficult for an adversary to fingerprint a deployed patch, QUICKSAND's patch selection process is built atop a moving target's defense-in-depth strategy combining cyber agility and honey-patching. In the context of software deception steering cyber maneuvers, we discuss the three components of this defense strategy [19]: configuration set C, timing function T, and movement strategy M.

4.1. Configuration Set

The effectiveness of software deception steering depends on the selection of a set of code versions (with honey-

```
Data: Π: patch set, B: base repository, HP: honey-patch
      repository
Result: set of conflict-free patch sets
1  cs ← ∅
2  Δ ← B
3  for (p1, p2) ∈ Π² do
4  begin
5      if ¬pull({p1, p2}, HP, Δ) then
6      begin
7          cs ← cs ∪ {(p1, p2)}
8      end
9      obliterate('[.*]-hp$', Δ)
10 end
11 remove(Δ)
12 return {S ∈ ℘(Π) | S² ∩ cs = ∅}
```

Figure 5: Conflict-free patch set generation algorithm

patches) that can be deployed at any point in time. This requires each code version to be *conflict-free*.

Conflict-free Code Versions. A conflict in our system is defined by the following syntactic rule: if (honey-)patch A and (honey-)patch B prescribe different contents for the same line of code, then A and B cannot coexist automatically in the same code version. A code version can be viewed as an element in the power-set of patches, i.e., $v \in \wp(\Pi)$. Thus, pruning this power set based on the pairwise definition of conflict between patches results in the configuration set C for the MTD.

Figure 5 details (in pseudocode) the algorithm for generating conflict-free code versions (or patch-sets) given the inputs Π representing the set of available security patches, the base repository B containing regular patches, and the repository HP of honey-patches. Lines 1–2 initializes the conflict-set cs to be empty, and a temporary repository Δ as a copy of B. The algorithm then populates cs with all conflicting patch pairs $\in \Pi \times \Pi$ (or Π^2) by checking the result of merging the corresponding honey-patch pair from HP into Δ and then resetting Δ between each merge operation (lines 3–10). Line 11 removes the temporary repository. Finally, in Line 12 the set of conflict-free patch sets is generated by pruning out all code versions from $\wp(\Pi)$ that contain conflict-pairs. While complex pruning rules (such as constraining honey-patches to be applicable only to releases officially reported in the Common Platform Enumerations database) can be specified, it reduces the cardinality of the configuration set and therefore the available options for the cyber maneuver.

4.2. Timing Function

QUICKSAND uses an event-based timing function T [19]. In this setting, when alerts are triggered by our system, we use it to compute the existence of a particular attacker type (discussed in the next section) and adapt our current deception strategy given this knowledge. In scenarios where alerts are ubiquitous, we consider a hybrid T that uses aggregate alert information over a time-period to modify the movement strategy.

4.3. Game-theoretic Movement Function

Given the set of conflict-free code versions, the system must decide which is to be deployed at the time of switching. To this end, we first consider a *game-theoretic modeling* of the interaction between an adversary and QUICKSAND. Then, we define an optimal deception selection strategy and describe methods to compute it.

In real-world settings, defenders often target adversaries having a particular set of characteristics. Curating the patch set selection strategy to the nuances of this adversarial profile thus results in more effective countermeasures. For example, it is ineffective to honey-patch only older, nearly obsolete vulnerabilities to gather threat intelligence about expert adversaries armed with the newest exploits. To address this concern, we model the problem as a two-player Bayesian Stackelberg Game (BSG) inspired by prior MTD web defense models [11].

Our BSG can be defined as a tuple $\langle \mathcal{D}, \mathcal{A}, A^{\mathcal{D}}, A^{\mathcal{A}}, U^{\mathcal{D}}, U^{\mathcal{A}}, P \rangle$, where $\mathcal{D}$ denotes the defender, $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_\theta\}$ denotes the θ types of attacker, $A^{\mathcal{D}}$ and $A^{\mathcal{A}} = \{A_1^{\mathcal{A}}, \ldots, A_\theta^{\mathcal{A}}\}$ denote the action-sets A of the players, $U^{\mathcal{D}} = \{U_1^{\mathcal{D}}, \ldots, U_\theta^{\mathcal{D}}\}$ and $U^{\mathcal{A}} = \{U_1^{\mathcal{A}}, \ldots, U_\theta^{\mathcal{A}}\}$ denote their utilities (with the subscripts representing the utility of the players corresponding to the adversary's type), and $P = \{P_1, \ldots, P_\theta\}$ denotes a probability distribution that represents the likelihood of facing each attacker type. Our goal is to derive a robust deception strategy that works well, in expectation, against all attacker types. Next we discuss how each of these model parameters are obtained and use real-world examples to elucidate the descriptions.

Players ($\mathcal{D}, \mathcal{A}$). The defender $\mathcal{D}$ represents the administrator who sets up the system, inspects reported alerts, and chooses to deploy a particular deception measure. The attacker $\mathcal{A}$ has three types according to skill set level—script kiddie ($\mathcal{A}_1$), early adopter ($\mathcal{A}_2$), and APT attacker ($\mathcal{A}_3$). As the names suggest, $\mathcal{A}_3$ is an expert who spends time in identifying vulnerabilities against a system, whereas $\mathcal{A}_1$ only uses attacks that have publicly available implementation of exploits, and $\mathcal{A}_2$ is biased towards exploits that are trending. A more formal distinction follows in our discussion of player action sets.

Actions (A). The defender's actions $A^{\mathcal{D}} = C = \{v_1, \ldots, v_n\}$ consist of the n feasible candidate patch-sets found using the algorithm in Figure 5. A defender can choose any of these actions (also referred to as a pure strategy) at any point in time. Given the patch sets used by the defender, we compile a list of known exploits E that can be used by an attacker. We enumerate the exploits against our system in Table 1 and consider subsets of E, using the `published` and the `public_exploit` fields in the metadata object of the exploit (see Listing 1), to describe the exploits available to each attacker type.

The script-kiddie's ($\mathcal{A}_1$) attack set consists of CVEs that have a known public exploit available. The early-adopter's ($\mathcal{A}_2$) attack set comprises vulnerabilities that have a public exploit available for which the published date is less than

Table 1: Summary of (honey-)patchable vulnerabilities with corresponding CVSS scores.

Vulnerability	Description	Software	CVSS		
			Impact (I)	Exploitability (E)	Overall (O)
CVE-2014-0160	Information leak	Openssl	5.4 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N/E:H/RL:O/RC:C/CR:H/IR:X/AR:X/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:H/MI:N/MA:N	3.7	8.9
CVE-2012-1823	System remote hijack	PHP	3.4 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:L/E:H/RL:O/RC:C/CR:M/IR:M/AR:M/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:L/MI:L/MA:L	3.7	7.0
CVE-2011-3368	Port scanning	Apache	1.4 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N/E:P/RL:O/RC:C/CR:M/IR:X/AR:X/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:L/MI:N/MA:N	3.5	4.8
CVE-2014-6271	System hijack	Bash	6.1 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H/E:H/RL:O/RC:C/CR:H/IR:H/AR:H/MAV:N/MAC:L/MPR:N/MUI:N/MS:C/MC:H/MI:H/MA:H	3.7	9.5
CVE-2014-6277	System hijack	Bash	6.1 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H/E:H/RL:O/RC:C/CR:H/IR:H/AR:H/MAV:N/MAC:L/MPR:N/MUI:N/MS:C/MC:H/MI:H/MA:H	3.7	9.5
CVE-2014-0224	Session hijack and information leak	Openssl	5.9 CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N/E:F/RL:O/RC:C/CR:H/IR:H/AR:X/MAV:N/MAC:H/MPR:N/MUI:N/MS:X/MC:H/MI:H/MA:X	2.1	7.5
CVE-2010-0740	DoS via NULL pointer dereference	Openssl	2.1 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L/E:P/RL:O/RC:C/CR:X/IR:X/AR:H/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:N/MI:N/MA:L	3.5	5.5
CVE-2010-1452	DoS via request that lacks a path	Apache	1.4 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L/E:P/RL:O/RC:C/CR:X/IR:X/AR:M/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:N/MI:N/MA:X	3.5	4.8
CVE-2016-6515	DoS via request that lacks a path	OpenSSH	5.4 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H/E:H/RL:O/RC:C/CR:X/IR:X/AR:H/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:N/MI:N/MA:H	3.7	8.9
CVE-2016-7054	DoS via heap buffer overflow	Openssl	5.4 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H/E:H/RL:O/RC:C/CR:X/IR:X/AR:H/MAV:N/MAC:L/MPR:N/MUI:N/MS:X/MC:X/MI:X/MA:H	3.7	8.9
CVE-2017-5941	System hijack	Node.js	4.0 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H/E:H/RL:O/RC:C/CR:L/IR:L/AR:L/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:H/MI:H/MA:H	3.7	7.6
CVE-2017-7494	System hijack	Samba	5.9 CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H/E:H/RL:O/RC:C/CR:H/IR:H/AR:L/MAV:N/MAC:L/MPR:N/MUI:N/MS:U/MC:H/MI:H/MA:H	3.7	9.4

$t = 5$ years. Thus, $\mathcal{A}_2$'s attack set includes the last two CVEs in the list shown in Table 1 (i.e. $|E_2| = 4$). The APT attacker ($\mathcal{A}_3$) can write exploits for any of the existing CVEs in the list, and has all the attack actions available (i.e. $|E_3| = |E| = 12$).

Utilities (U). To design the utility for the players, we primarily consider metrics that are a part of the Common Vulnerability Scoring System (CVSSv3) [20]. We first define a generic reward structure and discuss how it can capture the various aspects of cyber deception. Then, we highlight how we can obtain numeric values that represent the utility of the players.

The utility structure for a player, given that defender $\mathcal{D}$ deploys code version v and attacker $\mathcal{A}_i$ executes an exploit $e \in E_i$, is as follows:

$$U_i^{\mathcal{D}}(v,e) = \begin{cases} +r_i^{\mathcal{D}}(v,e) - c(v) & \text{if } e\text{-hp} \subset v \\ -I^{\mathcal{D}}(e) - c(v) & \text{otherwise} \end{cases}$$

$$U_i^{\mathcal{A}}(v,e) = \begin{cases} -r_i^{\mathcal{A}}(v,e) - c(e) & \text{if } e\text{-hp} \subset v \\ +I^{\mathcal{A}}(e) - c(e) & \text{otherwise} \end{cases}$$

In the first case, where the code version deployed has a honey-patch for exploit e that the attacker decides to exploit, the reward for the defender has two components. First, $\mathcal{D}$ gets a positive reward of $r_i^{\mathcal{D}}(v,e)$ because the attacker $\mathcal{A}_i$ was trapped using the honey-patch. This value is specific to the exploit being honey-patched as it needs to account for its intelligence-gathering worth (e.g., IPs used by the attacker) combined with the actionable protective measures that can be taken (e.g., add such IPs to the firewall deny-list). In this regard, given the attacks in our system, we consider the following ordering: $r_i^{\mathcal{D}}(v, \text{DoS}) \leq r_i^{\mathcal{D}}(v, \text{Port Scanning}) < r_i^{\mathcal{D}}(v, \text{Info Leakage}) < r_i^{\mathcal{D}}(v, \text{System Hijacking})$.

While the reward structure can encode context-specific information about the operating environment (e.g., value of targeted assets, mission critical requirements) or the attacks (e.g., targeted port-scanning, distributed *vs.* targeted DoS), we do not make such fine-grained distinctions as the production context isn't fully defined. This allows us to disregard the equality condition in the first inequality relating to the DoS and port-scanning vulnerabilities; we design uniformly spaced rewards in the range $[0, \max_e I^{\mathcal{D}}(e)]$. Second, the value $c(v)$ represents the cost of deploying a particular honey-patch on the Quality of Service (QoS) metrics on a system. We assume that all conflict-free patch sets v have the same cost (i.e. these values cannot incentivize $\mathcal{D}$ to pick a particular code version based on QoS metrics) but can consider a nuanced value for $c(v)$ when this distinction becomes necessary.

Attacker $\mathcal{A}_i$, when trapped by a honey-patch, incurs the cost of crafting and executing the exploit $c(e)$. This cost is dependent on the complexity of an attack (represented by its exploitability score) and the temporal metrics (what kind of an exploit or patch is available and how reliable the source is). Thus, we multiply the exploitability score (ES) of CVSS with the temporal metrics to obtain $c(e)$, similar to the way temporal scores are obtained using the CVSS's base

score (BS). The other negative reward $r_i^{\mathcal{A}}(v, e)$ captures $\mathcal{D}$ gaining knowledge of an attacker's TTPs. In our model, we assume that the attacker is unaware of which vulnerabilities are honey-patched, and thus $r_i^{\mathcal{A}}(v, e) = 0 \, \forall i \in \{1, \ldots, \theta\}$. One may choose to assign different scores to different players. For example, this can be used to reflect the fact that an APT attacker ($\mathcal{A}_3$) is better equipped to detect the deception.

In the second case, when v does not contain a honey-patch for e, the attacker can either gain reconnaissance if a regular patch is deployed by leveraging the attack failure information, or cause full impact without getting caught if no regular patches are available (for relatively new CVEs). For the latter case, $\mathcal{D}$ receives a negative utility against $\mathcal{A}_i$ with magnitude equal to the impact score, while $\mathcal{A}_i$ receives a positive utility with magnitude equal to the overall score. The overall score trades off the impact of the attack with the complexity of constructing and executing it. Table 1 shows the CVSSv3 metrics corresponding to the individual exploits of the formulated game, leveraged to calculate the utilities. For the former case, $\mathcal{D}$'s loss is a fraction of the impact score for giving out attacker info, whereas $\mathcal{A}_i$ considers its effort cost and the utility of gathering the information about patches.

Attacker Type Probabilities (P). We start with an initial probability distribution over attacker types (denoted as $\langle \Pr(\mathcal{A}_1), \ldots, \Pr(\mathcal{A}_\theta) \rangle$) that can be obtained by analysis of historical data (from similar systems) by security experts. Over repeated interactions, we can utilize an alert a raised by the analysis and correlation module to update the attacker type probabilities P_i as follows.

$$
\begin{aligned}
P_i &= \Pr(\mathcal{A}_i|a) = \Pr(\mathcal{A}_i) \cdot \Pr(a|\mathcal{A}_i) \\
&= \alpha \Pr(\mathcal{A}_i) \cdot \sum_e Pr(a|e) \cdot \Pr(e|\mathcal{A}_i) \\
&= \alpha \Pr(\mathcal{A}_i) \cdot \sum_e Pr(a|e) \cdot \mathbb{I}(e \in \mathcal{A}_i)
\end{aligned}
$$

where α represents the normalization factor and $\mathbb{I}$ represents an indicator function that equals 1 if the condition is met or 0 otherwise. The value $\Pr(e|\mathcal{A}_i)$ should ideally represent the strategy of an attacker type if they were to behave rationally. However, an alert may be observed for any of the available exploits, even when it is a sub-optimal choice for a rational adversary; we account for this irrationality by using the indicator function. Further, if an alert is generated by multiple exploit actions e, an attacker type with larger number of such exploits should be assigned higher probability. Lastly, while some alert systems, such as anomaly detection systems based on machine learning, may detect certain exploits imperfectly, we limit ourselves to deterministic detection mechanisms.

As an example, consider the use of a CVE from 2014 that is distinctive in observing a particular system alert. Given $\mathcal{A}_2$ cannot perform this attack action, P_2 becomes zero. This probability is distributed between P_3 and P_1, as $\mathcal{A}_1$ or $\mathcal{A}_3$ may have generated this alert. Thus, the initial distribution $\langle 0.4, 0.4, 0.2 \rangle$ becomes $\langle 0.67, 0, 0.33 \rangle$ after the first interaction with the attacker.

4.4. Strategy Computation

A strong threat model must account for an adversary capable of performing target reconnaissance. In game-theoretic terms, this boils down to the use of *Stackelberg Equilibrium*, which encodes the assumption that the defender acts as a leader while the attacker, who takes the role of a follower, is aware of the defender's deployment strategy [11, 21]. In this setting, the defender plays a mixed strategy (i.e., a probabilistic strategy over his actions) making it impossible for the attacker to fingerprint the deception strategy in place at any given point in time. In this Bayesian Stackelberg Game setting, we can calculate the optimal movement strategy ($\vec{x}$) for the defender by maximizing $\mathcal{D}$'s expected utility, as follows,

$$
\sum_i \sum_v \sum_e P_i \, x_v \, q_e^i \, U_i^{\mathcal{D}}(v, e) \tag{1}
$$

where each attacker $\mathcal{A}_i$'s strategy $\vec{q}^i$ is calculated with the knowledge of the defender's strategy $\vec{x}$ by maximizing $\sum_e \sum_v x_v \, q_e^i \, U_i^{\mathcal{A}}$, subjected to constraints that $\vec{q}^i$ represents a probability distribution. Both optimizations—maximizing the defender's expected utility given the attacker maximizing their utility—can be folded into a single DOBSS *mixed integer linear program* [10]. We use this formulation for calculating the strategy of version emulation in QUICKSAND. At the start of each time period, we use this mixed strategy to select a particular code version at random and proceed with its deployment.

Version Deployment. Upon completion of patch selection, QUICKSAND deploys a new version of the application into the target environment. Figure 2 outlines the steps taken to deploy an application. The first step consists of creating a *working* repository for the application, by first pulling all patches from `base` into `target`, and then pulling only the selected honey-patch subset into `target`. This yields a working repository state that is *tagged* with the selected application version. The final step consists of building the target application from sources, using a user-supplied configuration as supplemental input. The configuration parameters are specified per application, as shown in the configuration file illustrated in Listing 2, to set up the build environment and release the new application version.

5. Implementation

We developed an implementation of QUICKSAND for the 64-bit version of Linux. The implementation consists of four Python components: the *repository handler* module consists of about 150 lines of code and wraps Darcs CLI [17] to offer an API to access the version control system. The *analyzer* component consists of 90 lines of code, and leverages *py-idstools* [22] to parse IDS signature maps and events sourced in unified2 format (a serialized binary stream

Listing 2: QUICKSAND example configuration file

```
[Apache−1]
app = apache
base_repo = ../data/base
hp_repo = ../data/hp
deploy_repo = ../data/deploy
configure_command = make
install_command = make install
patches = CVE−2014−0160:CVE−2014−6271: \
      CVE−2014−6277:CVE−2014−7169: ...

[Apache−2]
app = apache
...

[OpenSSL]
app = openssl
...
```

format specification for IDS events), and *vFeed* [18] to fetch and aggregate threat metadata to alert objects. The *patch selector* module consists of an additional 140 lines of code, and the *version deployment* module adds about 80 lines of code to the system. Our implementation depends on a deployment environment that has been pre-configured with a honey-patching framework, along with its process sandboxing and monitoring facilities [2].

5.1. Conflict-free Patch Sets

Table 2 summarizes the conflict-free patch sets used as inputs to our game-theoretic decision process. These serve as candidate *versions* for patch selection, and encode information about affected software, such as application version compatibility. Each patch set implicitly encodes the availability of regular (e.g., CVE-2017-7494) and honey (e.g., CVE-2017-7494-hp) patch selections. Moreover, our patch repository maintains patch metadata that is used to filter *unpatchable* patch sets—patch selections for which a version deployment is infeasible due to patch compatibility and operation requirements.

5.2. Simulation Results

Table 3 highlights simulation results obtained using three different movement strategies: *static deception*, *uniform random strategy* (URS), and *Bayesian Stackelberg Game* (BSG). For simulation, we assume that the four latest vulnerabilities cannot be patched due to lack of officially available patches; the defender faces maximum impact for these and smaller impact for other vulnerabilities that are regularly patched due to leakage of reconnaissance information. The reward values for the defender shown in Table 3 are plotted across 12 different runs. In each run, we consider a distinct exploit is detected and update the beliefs over the attacker types accordingly.

In comparison to a static strategy that deploys the most profitable honey-patch set, software deception steering, regardless of employed movement strategy, increases system administrators' expected utility. Although, in our game setup, the use of uniform random strategy (URS) (i.e., selecting either of the 17 versions with equal probability at deployment time) is sub-optimal when compared to the Bayesian Stackelberg Equilibrium strategy. Further, the game-theoretic strategy identifies 6 code-versions devoid of security benefits, significantly reducing the defender's overhead of maintaining all 17 patch-sets. While the patch-sets #12 and #14 are pruned-out as #15 is a strict-subset, we also notice path-set #13 has a non-zero deployment probability. The patch-sets $\{1, 8, 9, 10, 12, 14\}$ are assigned zero-probability of deployment. Among the patch-sets that have non-zero probabilities of deployment (on average across the 12 runs), 5 of them have the highest deployment probability of 14.4% and the patch-set #7 has the lowest deployment probability of 1.1%.

While our game parameters are based on simulation results over rewards obtained from security databases, human-subject case studies must be conducted to understand the true benefits of this model. In the future, we plan to validate QUICKSAND under these three movement strategies in empirical attack-defense exercises.

Table 2: Summary of conflict-free patch versions (-hp implied)

#	Patch Set	Affected Software (CPE)
1	CVE-2014-0160	openssl:1.0.1f (≤)
2	CVE-2014-6271, CVE-2014-6277	bash:[4.3, 3.2.48, 2.0.5, 1.14.7] (≤)
3	CVE-2014-0160, CVE-2014-6271, CVE-2014-6277	openssl:1.0.1f (≤), bash:[4.3, 3.2.48, 2.0.5, 1.14.7] (≤)
4	CVE-2010-1452	http_server:2.2.15 (≤)
5	CVE-2011-3368	http_server:[2.2.21,2.0.64,1.3.68] (≤)
6	CVE-2010-1452, CVE-2011-3368	http_server:2.2.15 (≤)
7	CVE-2012-1823	php:[5.4.1, 5.3.10] (≤)
8	CVE-2016-6515	openssh:7.2 (≤)
9	CVE-2014-0224	openssl:[1.0.1f,1.0.0l,0.9.8y] (≤)
10	CVE-2014-0160, CVE-2014-0224	openssl:1.0.1f (≤)
11	CVE-2010-0740	openssl:0.9.8m (≤)
12	CVE-2010-0740, CVE-2014-0224	openssl:0.9.8m (≤)
13	CVE-2016-6515, CVE-2014-0224	openssh:7.2 (≤), openssl:1.0.1f (≤)
14	CVE-2016-6515, CVE-2010-0740	openssh:7.2 (≤), openssl:0.9.8m (≤)
15	CVE-2016-6515, CVE-2014-0224, CVE-2010-0740	openssh:7.2 (≤), openssl:0.9.8m (≤)
16	CVE-2017-5941	node-serialize:0.0.4 (≤)
17	CVE-2017-7494	samba:[4.1.23, 4.0.26,3.6.25,3.5.22] (≤)

Table 3: Benefits of different patch selection strategies.

Strategy	Expected Utility ↑	# Patch-sets used ↓
Static	-5.87	**1**
URS	-1.66 ± 0.81	17
BSG	$\mathbf{-1.26 \pm 0.89}$	11

6. Future Work

Experimental validation. QUICKSAND is an ongoing project, and future work is planned to fully evaluate our software version emulation strategy. We tested our approach on an experimental setup comprising virtual machines pre-configured for honey-patching [2]. Prior work has examined performance characteristics and the effectiveness of honey-patching [23, 14, 2] for cyberdeception.

We plan to empirically evaluate our approach based on a testing harness that streams the system synthetically generated, labeled attack data derived from real network traffic logs [24]. The labeled data will provide a ground truth to assign performance scores for each software-version generated by QUICKSAND, in a realistic and repeatable manner. Toward this goal, we plan to collect a pool of honey-patched vulnerabilities for highly-targeted server applications and libraries and craft exploit scripts to inject attacks into the regular traffic for the evaluation. Once we have gathered labeled data from our tests, we will extract features from the data set and tune QUICKSAND's version ranking function.

Human-subject evaluation. We also plan to leverage human-subject experimentation to evaluate our techniques. To this end, we are currently designing attack-defense capture-the-flag (CTF) exercises to assess the effectiveness of different deception configurations against human attackers. The goal is to use CTF data to derive realistic set points for our game-theoretic model.

7. Related Work

Cyber Agility. Moving target defenses (MTDs) [25] seek to thwart attacks by mutating or evolving digital environments faster than adversaries can adapt. MTD can be viewed as a subclass of the broader field of *cyber agility* [26], which includes any reasoned modification to a system or environment in response to a functional, performance, or security need. Such approaches sometimes benefit from deceptive ploys that can impede adversarial adaptation to the defense, but deception is not a requirement for agility or MTD to be effective—if the cyber-maneuver is faster than the enemy can react, the defense can be effective without any deceptive element.

MTD techniques can be broadly classified into *host-based* approaches, such as address space layout randomization [27, 28, 29], instruction set randomization [30], multi-variant execution environments [31], and *network-based* approaches, including address hopping [32, 33, 34], dynamic routes [35], and dynamic topology [36]. Using a typical attack kill chain (e.g., reconnaissance, access, development, launch, and persistence) as a taxonomy, the primary focus of host-based approaches is on development and launch phases, while network-based techniques focus primarily on the reconnaissance phase. This is useful for accommodating multiple threat models and designing evaluation strategies for each technique (e.g., overhead to perform reconnaissance and map a network topology).

Many of these agile defenses can benefit from deception (e.g., using network decoys to affect the perceived topology of an enterprise network to impede reconnaissance). Dually, good deceptions are often agile (i.e., their performance characteristics are indistinguishable from the target they impersonate), therefore creating a reverse synergy between such technologies. Our work benefits from research advances in MTD and extends the class of possible cyber maneuvers with a new mechanism based on software deception to inform the adaptation process.

Game-Theoretic Defense-in-Depth. MTD has been used to augment existing cyber systems with defense-in-depth. These efforts can be organized based on the cyber surfaces they move [19]. Our work introduces a new deceptive cyber maneuver for honey-patching that thwarts exploits on the exploration surface (e.g., probing) and on the attack surface (e.g., system hijacking). Heuristic movement strategies have been shown to be detrimental to the performance of these dynamic systems, leading us to take a principled approach that models the cyber-interaction as a game to create optimal movement strategies [37, 38, 21]. While we leverage existing security knowledge in publicly available databases (similar to [11]), we also consider contextual information and highlight important future research directions for developing a language-based approach and security metrics for cyber-deception. Unlike conventional MTD approaches that showcase effectiveness of toy domains and simulation environments [19, section IV], our work discusses a concrete architecture and a deployment model for enhancing enterprise security operations with cyber deception.

8. Conclusion

Our approach enhances vulnerability patching with a moving target defense technique that makes applications less predictable and more robust against attackers anti-deception efforts. Toward this end, we designed and implemented an adaptive, software version-emulation architecture in which the set of honey-patched vulnerabilities in a target application is dynamically re-selected to increase the likelihood of deceiving and entrapping attackers. Leveraging a game-theoretic analysis that automates and optimizes (honey-)patch management, our framework computes effective movement strategies based on contextual threat metadata and attacker model.

Acknowledgments

The research reported herein was supported in part by ONR award N00014-17-1-2995, and by U.S. ACC-APG / DARPA award W912CG-19-C-0003. Any opinions, recommendations, or conclusions expressed are those of the authors and not necessarily of the aforementioned supporters.

References

[1] Edgescan, "Vulnerability statistics report," January 2019.

[2] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[3] M. Musch, M. Härterich, and M. Johns, "Towards an automatic generation of low-interaction web application honeypots," in *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES)*, 2008.

[4] C. Leita, K. Mermoud, and M. C. Dacier, "ScriptGen: An automated script generation tool for Honeyd," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[5] D. Fraunholz, D. Reti, S. D. Anton, and H. D. Schotten, "Cloxy: A context-aware deception-as-a-service reverse proxy for web services," in *Proceedings of the 5th ACM Workshop on Moving Target Defense (MTD)*, 2018.

[6] W. A. Arbaugh, W. L. Fithen, and J. McHugh, "Windows of vulnerability: A case study analysis," *IEEE Computer*, vol. 33, no. 12, 2000.

[7] R. Lippmann, S. Webster, and D. Stetson, "The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection," in *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[8] E. Rescorla, "Is finding security holes a good idea?," *IEEE Security & Privacy*, vol. 3, no. 1, 2005.

[9] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen, "A trend analysis of exploitations," in *Proceedings of the 22nd IEEE Symposium on Security & Privacy (S&P)*, 2001.

[10] P. Paruchuri, J. P. Pearce, J. Marecki, M. Tambe, F. Ordonez, and S. Kraus, "Playing games for security: An efficient exact algorithm for solving Bayesian Stackelberg games," in *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008.

[11] S. Sengupta, S. G. Vadlamudi, S. Kambhampati, A. Doupé, Z. Zhao, M. Taguinod, and G.-J. Ahn, "A game theoretic approach to strategy generation for moving target defense in web applications.," in *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2017.

[12] NIST, "Vulnerability summary for CVE-2014-6277," 2014.

[13] D. Roundy, "Darcs: Distributed version management in Haskell," in *Proceedings of the ACM SIGPLAN Workshop on Haskell*, 2005.

[14] F. Araujo, M. Shapouri, S. Pandey, and K. Hamlen, "Experiences with honey-patching in active cyber security education," in *Proceedings of the 8th Workshop on Cyber Security Experimentation and Test (CSET)*, 2015.

[15] F. Araujo and K. W. Hamlen, "Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception," in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[16] I. Lynagh, "An algebra of patches," tech. rep., University of Oxford, 2006.

[17] Darcs, "Darcs version control system," 2016.

[18] Toolswatch, "vFeed – The correlated vulnerability and threat database," 2016.

[19] S. Sengupta, A. Chowdhary, A. Sabur, A. Alshamrani, D. Huang, and S. Kambhampati, "A survey of moving target defenses for network security," *IEEE Communications Surveys & Tutorials*, 2020.

[20] FIRST, "Common vulnerability scoring system." www.first.org/cvss, June 2019.

[21] S. Sengupta, A. Chowdhary, D. Huang, and S. Kambhampati, "Moving target defense for the placement of intrusion detection systems in the cloud," in *International Conference on Decision and Game Theory for Security (GameSec)*, 2018.

[22] Py-idstools, "Py-idstools Python library," 2016.

[23] X. Han, N. Kheir, and D. Balzarotti, "Evaluation of deception-based web attacks detection," in *Proceedings of the Workshop on Moving Target Defense*, 2017.

[24] G. Ayoade, F. Araujo, K. Al-Naami, A. M. Mustafa, Y. Gao, K. W. Hamlen, and L. Khan, "Automating cyberdeception evaluation with deep learning," in *Proceedings of the 53rd Hawaii International Conference on System Sciences (HICSS)*, 2020.

[25] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54. Springer Science & Business Media, 2011.

[26] P. McDaniel, T. Jaeger, T. F. La Porta, N. Papernot, R. J. Walls, A. Kott, L. Marvel, A. Swami, P. Mohapatra, S. V. Krishnamurthy, and I. Neamtiu, "Security and science of agility," in *Proceedings of the 1st ACM Workshop on Moving Target Defense (MTD)*, 2014.

[27] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[28] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[29] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan, "Preventing overflow attacks by memory randomization," in *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2010.

[30] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[31] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, 2009.

[32] E. Al-Shaer, "Toward network configuration randomization for moving target defense," in *Moving Target Defense* (S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, eds.), Springer, 2011.

[33] T. E. Carroll, M. Crouse, E. W. Fulp, and K. S. Berenhaut, "Analysis of network address shuffling as a moving target defense," in *Proceedings of the IEEE International Conference on Communications (ICC)*, 2014.

[34] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: Transparent moving target defense using software defined networking," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[35] S. T. Trassare, R. Beverly, and D. Alderson, "A technique for network topology deception," in *Proceedings of the IEEE Military Communications Conference (MILCOM)*, 2013.

[36] P. Kampanakis, H. Perros, and T. Beyene, "SDN-based solutions for moving target defense network protection," in *Proceedings of the IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2014.

[37] K. M. Carter, J. F. Riordan, and H. Okhravi, "A game theoretic approach to strategy determination for dynamic platform defenses," in *Proceedings of the ACM Workshop on Moving Target Defense (MTD)*, 2014.

[38] A. Clark, K. Sun, L. Bushnell, and R. Poovendran, "A game-theoretic approach to IP address randomization in decoy-based cyber defense," in *International Conference on Decision and Game Theory for Security (GameSec)*, 2015.