Automated benchmark network diversification for realistic attack simulation with application to moving target defense

Alexander Bajic¹ · Georg T. Becker²

Published online: 31 May 2021
© The Author(s) 2021

Abstract

With numbers of exploitable vulnerabilities and attacks on networks constantly increasing, it is important to employ defensive techniques to protect one's systems. A wide range of defenses are available and new paradigms such as Moving Target Defense (MTD) rise in popularity. But to make informed decisions on which defenses to implement, it is necessary to evaluate their effectiveness first. In many cases, the full impact these techniques have on security is not well understood yet. In this paper we propose network defense evaluation based on detailed attack simulation. Using a flexible modeling language, networks, attacks, and defenses are described in high detail, yielding a fine-grained scenario definition. Based on this, an automated instantiator generates a wide range of realistic benchmark networks. These serve to perform simulations, allowing to evaluate the security impact of different defenses, both quantitatively and qualitatively. A case study based on a mid-sized corporate network scenario and different Moving Target Defenses illustrates the usefulness of this approach. Results show that virtual machine migration, a frequently suggested MTD technique, more often degrades than improves security. Hence, we argue that evaluation based on realistic attack simulation is a qualified approach to examine and verify claims of newly proposed defense techniques.

Keywords Attack simulation · Network security · Moving target defense · Defense evaluation

1 Introduction

Evaluating and especially comparing the effectiveness of defensive techniques is important in order to decide upon which of them to implement to protect one's networks and systems. However, doing so is very difficult due to lacking metrics and norms. The ever-growing number of proposed defenses arising from newly introduced paradigms further amplifies the need for evaluation and comparison. Moving Target Defense is one of these paradigms, advocating proactive defense through mutating a system's appearance, thus repeatedly invalidating knowledge an attacker may have previously acquired. In consequence, attacks that rely on such outdated knowledge may fail, forcing the attacker to redo reconnaissance over and over again. This, in turn, is not only

time-consuming and thus costly, but also risky as it increases chances of being detected. As a result, the typical information asymmetry between an attacker who may take as much time as needed to prepare an attack, and the defender who must be prepared at all times, shifts in favor of the defender, which is why MTD is frequently introduced as a “game changer” in IT security [12,16,18,43]. Frequently cited MTD techniques are, for example, IP address randomization [11,20,26,28,33], suggesting to repeatedly alter addresses of connected nodes, as well as virtual machine (VM) migration [1,3,5,18,24], a technique that has been proposed in different forms and suggests repeated relocation of VMs across hypervisors to move them out of the attacker's reach. A comprehensive overview of MTD techniques can be found in the recent survey from Cho et al. [13]. While these techniques often come with convincing examples that emphasize their ability to improve security, the question on how to verify and especially quantify and fairly compare these security improvements remains unanswered.

In existing MTD literature, a lot of work focuses on developing a theoretical foundation to analyzing MTD techniques, with many approaches based on game theory [1,2,27,36]. Yet,

✉ Alexander Bajic
alexander.bajic@fu-berlin.de

Georg T. Becker
georg.becker@ruhr-uni-bochum.de

¹ Digital Society Institute, ESMT Berlin, Berlin, Germany

² DCSO, Berlin, Germany

these mostly intend to optimize attack and defense strategies or find equilibria in the presence of actors of certain capability. While this is ultimately needed to appropriately utilize respective techniques, it requires a defensive technique's effects to be fully understood and included in analysis. Otherwise, resulting strategies will not yield the intended effect on security when being applied in the real world for neglecting relevant aspects. Still, it appears that the effects of techniques are more often assumed than tested and generally perceived to be beneficial. The only downsides occasionally taken into account are potential service degradation or additional resource consumption that impact the cost functions of said evaluation approaches. Negative effects on security are not considered. Furthermore, analytical approaches to MTD evaluation frequently rely on formalization of reoccurring defense effects that oppose certain attack actions [16,18], which is generally feasible for active defenses. Static defense concepts such as firewalls or micro segmentation, however, cannot be represented that way and can therefore not be included in the analytical model, thus impeding comparability.

Other approaches only consider very specific threats or scenarios. Repeatedly randomizing IP addresses, for example, has been deemed useful based on the low probability for an attacker to correctly guess one or more IP addresses in a row [30]. While this is not far-fetched, in practice, an attacker has means and information at hand that make acquisition of IP addresses more efficient than guessing from a pool of possible addresses. Consequently, only looking at the guessing entropy is not enough to evaluate this defense. In addition, guessing entropy may not be suitable for other defenses such as VM migration, where determining the new location of a virtual machine is only one step towards a successful attack. Therefore, such defense-specific metrics are not applicable to compare different defenses.

In this paper we address these problems using attack and defense simulation. The idea is to model networks and attacks to simulate interaction and observe how far an attacker can infiltrate the network with different defenses enabled. The further an attacker gets, the more revenue he or she accumulates which is calculated from assigned values of compromised resources. This revenue may vary across defenses, thus ultimately allowing to compare them. Yet, to obtain meaningful results and determine a defense technique's real effects instead of just assuming them, sufficiently detailed modeling is required to make for realistic scenarios and actions. Furthermore, results on defense performance from simulation on a single benchmark network are not representative for a given defensive technique. Seemingly unimportant details of the underlying network model may have a significant impact on simulation development. This was highlighted in previous work where the starting location of VMs drastically impacted the defense performance, rais-

ing strong doubts on the usefulness of analyzing defenses in only few benchmark networks [8]. Therefore, we do not perform attack and defense simulation on a single modeled network, but on a large number of diverse benchmark networks that are carefully composed in an automated fashion based on a scenario definition and a "benchmark fuzzing" approach. This allows for quantitative analysis and comparison which was previously not possible and increases the chances of detecting corner cases. To put findings into perspective, the same defenses as in prior research [8] are investigated. These comprise IP address randomization [11,20,26,28,33] (short: *IP shuffling*), state-preserving VM migration [1,5,24] which we denote as *live migration*, state-resetting VM migration [3,18] denoted as *cold migration*, as well as sole *VM resetting*. The first three are frequently proposed and discussed MTD techniques for which even prototypical implementations have been suggested. Sole *VM resetting* [10] is not a Moving Target Defense in the classical sense for lacking the movement property. Yet, it is the main difference between *live* and *cold migration* so that investigating how effective this technique is on its own, poses an interesting question. Detailed descriptions of these techniques are given in the case study in Sect. 3.1.

1.1 Main contribution

This paper builds upon the 2018 NordSec conference paper [8] which showed the general feasibility of attack simulation and its ability to generate meaningful insights. However, it also showed how susceptible results are to minute details of the benchmark network. By simply modifying the starting location of VMs, the defense performance of VM migration changed drastically. The key findings of the previous work were:

1. Feasibility of detailed modeling and subsequent attack and defense simulation to generate meaningful insights on defense effects.
2. Revelation of security degrading effects of a frequently proposed proactive defense that had been deemed useful by other research so far.
3. Demonstrating the impact of minute and frequently neglected details on evaluation results, raising awareness that incidental evidence is not sufficient to evaluate defenses.

The lesson learned was that simulation is useful, yet basing it on only few benchmark networks is not sufficient to fairly investigate and compare defense techniques for missing the potentially wide range of effects that shed a different light on defenses.

To this end, this work presents a complete framework that lifts attack and defense simulation to the next level. Using

a sophisticated modeling language supporting a wide set of features to randomize and scale properties of generated benchmark networks, diverse instances that exhibit a high level of detail and realism can be generated automatically. A scheme we coined “benchmark fuzzing”. This allows to scale up simulation and reveal numerous additional effects of defenses that occur with varying frequencies, providing a broader understanding of the different defenses’ utility depending on environmental conditions. Additionally, new metrics are introduced to aggregate data and present findings in a comprehensible way, enabling a fair and comprehensive quantitative analysis of defenses. The contribution of this work can be summarized as follows:

1. Introducing benchmark fuzzing to automate generation of realistic benchmark networks in large numbers as opposed to previous manual modeling, thus minimizing effort while increasing test coverage.
2. Increasing size and complexity of investigated networks, extending the range of observed effects caused by different MTD techniques (*IP shuffling*, *live migration*, *cold migration*, and *VM resetting*).
3. Demonstrating applicability and usefulness of this approach in a case study analyzing 200 automatically generated benchmark networks, instead of two, to determine dominant effects of defenses and distinguish them from corner cases.
4. Defining new metrics to aggregate large quantities of simulation results into compact performance indicators instead of per-scenario analysis, thus supporting informed decision making.
5. Presenting findings that paint a differentiated picture of defense performance across diverse networks, raising strong concerns about the usefulness of VM migration as a security measure, thus contradicting other studies in this area. Results highlight the need to also consider negative effects when analyzing defenses. An aspect that has not gained the required attention in MTD literature yet.

1.2 Outline

The next section presents the attack simulation framework, giving a short introduction to its primary components. Section 3 presents a case study, showing how a specific experiment is implemented using our framework, with corresponding results presented in Sect. 4. Following this, numbers on performance and scalability of both benchmark network generation and simulation are provided in Sect. 5. Related work with a special focus on approaches to evaluation that came to other conclusions than ours, is presented in Sect. 6. Finally, the paper is concluded in Sect. 7, including an outlook to possible future work.

2 Attack modeling and simulation framework

The main purpose of the framework is to enable building realistic benchmark networks and simulating attacks on these in the presence of different defenses. This is done in a time-discrete round-based fashion with each round allowing the initiation of different attack and defense actions, depending on the fulfillment of their specific requirements. All actions take individual numbers of rounds to execute, have different success probabilities, and may alter the system state. The effectiveness of a specific defense is then evaluated by performing simulations with and without said defense enabled and comparing the differences in maximum attacker revenue, as well as the time it took to accumulate it. This revenue, in turn, is generated through compromising resources that are spread across the network and hold respective values. To be able to realistically analyze and compare different defense techniques, the following requirements must be met:

- *Realistic attacker capabilities* Attackers must be able to realistically interact with the system. They should not only be able to exploit vulnerabilities but also use legitimate functions and employ acquired knowledge. Furthermore, the requirements for attacker actions need to include sufficient detail to be realistic.
- *Realistic defense modeling* Similarly, defender actions need to be modeled accurately. That is, all state changes caused by a defense must be considered and not only intended beneficial ones. Otherwise, security-relevant effects may go unnoticed.
- *A wide range of benchmark networks* The performance of attacks and defenses may depend on environmental factors. Hence, to realistically compare defense techniques, the analysis should not be based on a single network configuration but on a wide range of configurations.
- *Common and quantifiable metrics* Resulting metrics should be uniform across all considered defenses without preferring one defense over the other.

2.1 Framework overview

Figure 1 depicts an overview of the modeling and simulation framework. The *Instantiator* uses the *Component Library*, containing ready-made definitions of commonly used elements, to automatically generate an arbitrary number of benchmark networks according to the *Scenario Definition*. Attacks are then simulated for each combination of network and defense, using the *Simulation Engine* that operates according to parameters from a *Config File*. Respective results are stored in *log files* representing the revenue earned by the attacker per round, as well as performed attacker and

Fig. 1 Overview of the attack simulation framework

defender actions. Once simulation is done, the *end state* is also stored in case further investigation is required.

Both *Scenario Definition* and *Component Library* are written in our modeling language that offers useful features enabling the definition of probabilistic and/or conditional scenarios. For example, numbers of clients, the chances for vulnerabilities to exist, or the size of subnets can be controlled via parameters that might either be fixed, subject to probability, or dependent on other conditions. Such parameters are included in the *Scenario Definition* and enable the generation of many different network instances, whose (dis)similarity depends on how extensively these diversifying features are used. Each instance is a full state description, defining every element in the network, including attackers and defenders. From a technical perspective, these instances are fully functional Prolog databases that are used by the *Simulation Engine* to simulate attack and defense and apply state changes accordingly. Consequently, for this purpose, the simulator relies on Prolog to perform deductive reasoning and determine the respective actors' options. A *Simulation Config* file is used to configure the simulation regarding active defenses, attack time and related options. In the following, involved components will be explained in more detail with regard to their purpose and functionality. A more technical description of how the benchmark network generator and the simulation engine are implemented, can be found in Appendix A.

2.2 Modeling language

While relying on Prolog is convenient to determine the different actors' options, as it allows to simply query for feasible actions, manually crafting extensive Prolog databases that incorporate the required level of detail is cumbersome and anything but intuitive. Ou et al. [35], who developed the attack graph generation tool MulVal which also relies on Prolog came to a similar conclusion, introducing means to mitigate the effort and pitfalls of manually modeling networks. Yet, the MulVal approach differs from the one presented here in two fundamental aspects. First, MulVal does not perform interactive simulation but derives attack trees from the attacker's perspective that do not incorporate dynamic defense. Second, it obtains networks to perform deductive reasoning from active network scanning. This

```

1 init (NAME, SUBNET, IPADDRESS,
  ARCHITECTURE, VULNS);
2
3 string name=$NAME$;
4 string cpu=$ARCHITECTURE$;
5 subnet belongsToSubnet=$SUBNET$;
6 string nodeType=$SUBNET$.type;
7 ipaddress tmpIp=new ipaddress("
  $NAME$. $SUBNET$", $IPADDRESS$);
8 interface myInterfaces=new interface
  ($SELF$.tmpIp);
9 string[] hasVuln=$VULNS$;
  
```

Listing 1 Template definition for elements of type node

enables users to create representations of real networks, which is convenient to assess the security of a specific setup. However, it requires the user to have such a network to begin with. Furthermore, when investigating the range of potential effects a defensive technique may have, and not only its impact on security in a specific case, a multitude of networks is needed, which is why our approach is based on synthesizing networks with an easy to use modeling language. The language's syntax is similar to that of well-known object-oriented programming (OOP) languages. It allows to freely describe the state of networked systems, as well as attacker and defender actions. Resulting definitions are more compact and easier to comprehend, yet can be automatically converted into Prolog facts and rules. To keep scenario definitions maintainable and improve usability, scenario definitions are modularly composed of re-usable code, allowing to build large and realistic scenarios with comparatively few instructions. For this purpose, templates can be defined that serve as blue prints to simplify the generation of numerous similar elements. Providing such a template definition once, it can be used whenever needed with a single line of code to create extensive elements with all their attributes. This is similar to the concept of classes in OOP.

Listing 1 depicts a template definition for elements of type *node*. For the sake of comprehensibility, all lines are covered step by step. Every template definition begins with the *init*-statement as shown in line 1. This statement serves to specify how values that are passed during the instantiation call are addressed to use them throughout the template and are always written in all capitals. In the given example, these are the node's name, the subnet it is supposed to belong to, its IP address, CPU architecture and potential vulnerabilities. The values referenced by those placeholders can subsequently be accessed by enclosing the placeholder in dollar signs where needed. Line 3 makes use of the passed value referenced through *\$NAME\$* when declaring and initializing the element attribute *name* that is of type *string*. Note that all statements are terminated with a semicolon. Similarly, in line 4, the element attribute *cpu* is defined which is also of

type *string*. While exhibiting the same syntax, line 5 introduces another concept. Instead of declaring a simple type such as *string*, *int*, or *boolean*, the specified type is *subnet* with the assigned value being whatever is referenced through *\$\$SUBNET\$*. Anything that is not a *string*, *int*, or *boolean* is assumed to be an element which may have attributes of its own. Consequently, the value being assigned here is not a string but a reference to an element. The statement in line 6 builds upon this concept by initializing the attribute *node-Type* of type *string* with a value obtained through accessing an attribute of the element referenced through *\$\$SUBNET\$*. Note that the respective element's attribute is directly pointed to by concatenating the element reference and the attribute's name with a dot (i.e. *\$\$SUBNET\$.type*).

Apart from assigning existing elements to attributes, as depicted in line 5, new ones can be created, as well. In line 7, an element of type *ipaddress* is instantiated, calling the respective template using the *new*-statement and passing it two values. The first one being a composition of *\$NAME\$* and *\$\$SUBNET\$* serving as a DNS name, and the second one being the IP address that the node template itself got during instantiation. The quotation marks that enclose the first value ensure that the composition of *\$NAME\$* and *\$\$SUBNET\$* is treated as a string. This is needed since the value of *\$\$SUBNET\$* is an element's unique identifier, primarily serving as a reference to said element, that is now simply being used to create a unique name. Such instantiation calls are always assembled in the same way. First, the type (*ipaddress*) and freely chosen name (*tmplp*) are specified, followed by an equal sign and the *new*-statement, specifying the template to be called. Line 8 represents the same concept of creating a new element serving as an attribute of the current element. In this case, the node is equipped with an *interface* that gets the newly created *ipaddress* element as a parameter to configure it. Finally, line 9 depicts the assignment of a *string* list as an attribute of the element. This is indicated by the square brackets that succeed the type specification. In such cases, the value being assigned must exhibit the format of a list, even if consisting of only one item (e.g. [*"vulnerability1"*]). Upon assigning more than one value, these are separated with commas.

To get an impression of how templates are used in practice, Listing 2 depicts a minimal scenario definition where two nodes are instantiated using the previously outlined template and equipped with operating systems. Again, statements are covered line by line. The scenario definition starts with instantiating an element of type *subnet* referred to as *intranet*, using the respective template and passing it three values. These are the subnet's name, followed by the type of nodes it contains, and its location. Please note that deciding which and how many values or variables to pass is up to the user designing the template. The three values from this example reflect the information that was used to describe subnets

```

1 subnet intranet=new subnet("
  networkName", "client", "local");
2
3 node host1=new node("myDesktop",
  intranet, "192.168.0.1", "x86", [
    meltdown"]);
4 os host1.os=new os("windows", "client",
  ["smb", "rdp"]);
5
6 node host2=new node("myLaptop",
  intranet, "192.168.0.2", "x86", [
    meltdown, "spectre"]);
7 os host2.os=new os("windows", "client",
  ["smb"]);

```

Listing 2 Minimal scenario definition to create 2 nodes and corresponding operating systems

in our experiments. Lines 3 to 7 span the instantiation of the two nodes and corresponding operating systems. The first one of these creates a complete node by calling the node template from before. Further, one can see how this instantiation call passes the previously discussed parameters, serving to customize the element to be created. Note that the last parameter exhibits the shape of a list as is required by the template. The next statement takes care of equipping the created node with an operating system. This is done by directly instantiating an *os* element from a template to be an attribute of *host1* that is referenced through *host1.os*. Lines 6 and 7 repeat this to create another node with deviating properties.

Though short, the two snippets present working code to define templates and use them in the course of a scenario definition, while introducing the fundamental concepts of how modeling with the language works. Furthermore, the example shows how using templates reduces complexity of the scenario definition as opposed to repeatedly describing elements in full.

2.3 Benchmark fuzzing

Benchmark fuzzing is the automated generation and diversification of realistic benchmark networks that serve as inputs for simulation-based evaluation. As our previous research indicates, minute variations in benchmark networks can have significant effects so that performing simulation on diverse networks is ultimately needed to understand how defenses and attacks perform in the presence of different environmental conditions. Benchmark fuzzing takes the burden of manually modeling such networks from the user, thus reducing effort while extending the spectrum of incorporated variations. In that sense, it resembles fuzzing known from software testing, where diverse inputs are generated automatically to test the behavior of software upon processing such inputs. While fuzz testing is not limited to generating realistic inputs only, but equally comprises invalid and unexpected

input, the proposed benchmark fuzzing scheme borrows the fundamental concept of automatically randomizing inputs.

To accommodate this, the formerly descriptive modeling language is extended with another layer of abstraction, allowing for a procedural definition of non-deterministic scenarios. Instead of specifying all aspects, scenarios may exhibit degrees of freedom regarding selected parameters, based on which diverse benchmark networks are generated. This is implemented through features such as *loops*, *if*-clauses, a functor to randomize selection of specific options, as well as *runtime variables* which can be used during instantiation. Resulting benefits include:

- omitting excessive repetitions with loops
- using conditions to switch between various alternatives within a scenario
- using runtime variables to store and retrieve information at any point during scenario generation
- maintaining and extending lists of elements for subsequent processing (e.g. group items that must be accessed and altered on particular occasions)
- employing probability to diversify single attributes and large parts of a scenario alike, to create differently shaped benchmark networks

What this means in practice is best illustrated with another example. For this purpose, we augment the scenario definition from Listing 2 with those newly added features to describe their effects. This adapted scenario definition is depicted in Listing 3.

Again, the scenario definition starts with instantiating an element of type *subnet* referred to as *intranet*. Lines 2 and 3 prepare runtime variables that serve as the prefix for IP addresses as well as a counter to automatically increment addresses. Runtime variables are marked as such by starting and ending with dollar signs. Lines 4 to 18 span the main loop to create nodes and corresponding operating systems with the loop's head specifying to repeat enclosed statements exactly 10 times, using runtime variable *\$I\$* to sequentially assume the values 1 to 10. Lines 5 and 6 assemble an IP address for the next node to be created and store it as *\$curIp\$*. Note that the symbol *+=* is used in line 6 to not overwrite, but extend the current value, so that a complete address is formed. Yet another runtime variable named *\$specVulns\$* is initialized in line 7 that will be used later on to specify each node's vulnerabilities. Being set to *none*, these nodes would not exhibit any vulnerabilities. However, the subsequent lines show how probabilistic decision making and conditional processing may affect such characteristics, starting with line 8. There, the variable *\$noOfVulns\$* obtains a value resulting from the *prob*-functor that is provided with two lists, the first one specifying the potential values to choose from and the second one specifying the chances at which these may be

```

1 subnet intranet=new subnet("intranet
  ", "client", "local");
2 $prefix$="192.168.0.";
3 $counter$=1;
4 for($I$ in range(1,10)){
5   $curIp$=$prefix$;
6   $curIp$+= $counter$;
7   $specVulns$=none;
8   $noOfVulns$=$PROB
    ([0,1,2],[50,25,25])$;
9   if($noOfVulns$ == 1){
10    $specVulns$=[ "meltdown" ];
11  };
12  if($noOfVulns$ == 2){
13    $specVulns$=[ "meltdown", "spectre"
14  ];
15  };
16  node host$I$=new node("pc$I$",
    intranet, $curIp$, "x86", $specVulns$);
17  os host$I$.os=new os("windows", "
    client", ["smb", "rdp"]);
18  $counter$+=1;

```

Listing 3 Scenario definition employing new features to create 10 nodes and corresponding operating systems

picked. For the statement in line 8 this translates to a 50% chance of picking 0 and a 25% chance of picking either 1 or 2. If the second list to specify probabilities is omitted, the framework will assume equal chances for all possible instances.

Using *if*-clauses to check the shape of *\$noOfVulns\$*, lines 9 to 14 change the contents of *\$specVulns\$* in different ways to contain either one or two vulnerabilities. In case the *prob*-functor returned 0, nothing happens and *\$specVulns\$* remains as *none*. Following these conditional statements, a *node* element is instantiated through calling the known *node* template. Note how the incrementing variable *\$I\$* is used for the node's identifier. Since these must be unique to unambiguously identify created elements, using *\$I\$* is convenient when creating elements in a loop. Further, one can see that the parameters being passed to the template to customize the element during instantiation may not be explicitly given anymore, but be represented through runtime variables, one being the probabilistically determined list of vulnerabilities. Line 16 then instantiates the corresponding node's OS in the same way as before, only differing in how the node is referenced (i.e. *host\$I\$*). Lastly, before the loop's end, the variable named *\$counter\$* is incremented manually to ensure that the next node is instantiated with a new IP address. For this purpose, the symbol *+=* is used again. Yet, this time not being applied to a string, as was the case with *\$prefix\$* in line 2, but a numeral, causing its value to be increased by 1.

Listing 3 only provides a small example, yet it shows how features introduced for benchmark fuzzing enable the procedural generation of probabilistic and conditional bench-

```

1 privilegeEscalation(os OS, actor
  ACTOR)
2 {
3     ACTOR.type=="attacker" &&
4     isInList("privEscVuln", OS.
  hasVuln) &&
5     NODE.os==OS &&
6     isInList(RCE, NODE.rceRights) &&
7     ! ( RCE.level==OS )
8
9     set application RCE.level=OS
10 };

```

Listing 4 Example definition of a generic privilege escalation

mark networks. As a result, creating numerous different, yet equally realistic instances requires considerably less effort. Section 3 provides insights on how we used these new features to create a wide set of benchmark networks. Interested readers are referred to Online Resource 1, containing the full scenario definition of the case study from Sect. 3, where presented features are used extensively.

2.4 Modeling attacks, defenses, and legitimate actions

Apart from describing the state of networks with all their components, the modeling language allows to define functions that represent attacks, defenses, as well as legitimate actions to manipulate the modeled system's state, thus enabling interaction and progression of the simulation. To keep this simple, functions are implemented as sets of conditions and effects, with the effects only being applied to the system state if related conditions are met. In that sense, functions are not imperative but declarative, meaning that they do not provide instructions on what to do, but specify the shape of valid targets and the desired target state, similar to how SQL-queries work. Consequently, the different types of conditions supported by the language serve to enable users to efficiently narrow down the set of targets that a certain function is applicable to. Effects, in turn, cover insertion, alteration and removal of state information. However, defining effects is optional so that functions may only contain conditions and, instead of causing any effects to the state model, will only return either *true* or *false*, depending on whether or not conditions are fulfilled. Such functions may be used as conditions themselves that are evaluated in the course of other functions. This way, frequently needed conditions can easily be grouped and reused.

As an example, Listing 4 depicts the “privilegeEscalation” function that has two input parameters, the target *OS* and the calling *ACTOR* as specified in the function head in line 1. These parameters are specified with their type (lowercase letters) and variable names (capital letters) through which

the values they hold are referenced throughout the function's body. Lines 3 to 8 cover the aforementioned conditions, starting with a check whether the actor element referenced through *ACTOR* is an attacker, by comparing its attribute *type* with a defined string. Such checks are needed since the simulator is fully agnostic to the different roles actors may have in the course of attack and defense simulation. In consequence, functions that should only be conducted by certain types of actors must be specific in this regard. Following this, the next condition in line 4 verifies that the referenced operating system *OS* exhibits the vulnerability needed to perform a privilege escalation as intended by this function. This is done using the *isInList*-functor which checks whether or not the list of values referenced through *hasVuln* contains the specified string “privEscVuln”. Remember that Listing 1 introduced *hasVuln* as a list of strings. The following conditions are specifically noteworthy as they do not contain any explicit values anymore, but employ additional variables. Line 5 introduces the variable *NODE* whose value is not specified but resolved through comparing which element has an attribute named *os* that specifically references the *OS* element that the function obtained as a parameter, emphasizing the declarative nature of function definitions. Assuming that this condition leads to a specific element to be referenced through *NODE*, the condition in line 6 serves to resolve yet another element by checking if *NODE* has a list attribute named *rceRights*, from which the variable *RCE* may be initialized. In the context of this model, *RCE* stands for “remote code execution”, representing an actors ability to exert control over a system, either on user or system level. Consequently, checking for the presence of elements in the attribute *rceRights* serves to determine if the given node is already under control of the actor. If this is the case, testing of conditions proceeds with verifying that this remote code execution privilege does not already yield system level privileges through checking if the *RCE*'s respective level is set to the *OS* and negating the whole condition with a preceding exclamation mark (!). Since all conditions must be fulfilled, they are combined through logical *AND*-operators represented through the double ampersand. Logical *OR*-operators, if needed, are represented through double pipes. Finally, in line 9, the effect is defined. In this case, this is as simple as setting the level of the previously resolved element *RCE* to *OS*, meaning that the actor obtained system level privileges. Note that in between the *set*-instruction and the attribute to change, a type *application* is given. This is due to the fact that elements referenced through the *level* attribute of an *rceRight* element are treated as type *application*. Apart from setting specific attributes, the effects of a function comprise adding (*add*) and deleting (*del*) attributes, as well as creating (*create*) new elements, which can be freely combined to define sophisticated target states.

```

1 liveMigration(node TARGETVM, actor
  ACTOR)
2 {
3   ACTOR.type=="defender" &&
4   TARGETVM.canBeMigrated=="true" &&
5   isInList(TARGETVM, OLDXEN.vms) &&
6   MIGRATIONPOOL=findAllWhere(X, (
7     isInList("xen", X.type) && !
8     isInList(TARGETVM, X.vms))) &&
9   NEWXEN=pickFromList(MIGRATIONPOOL)
10  &&
11  isInList(NEWXEN, DESHOST.os) &&
12  DESHOST.belongsToSubnet==DESTSUBNET
13  }
14 ;

```

Listing 5 Implementation of VM *live migration*

The given example covered a simple exploit to escalate privileges in the presence of a corresponding vulnerability. Defenses and legitimate actions are modeled the same way, merely differing in their specific conditions and effects. To illustrate this, Listing 5 depicts an implementation of VM *live migration*. As can be seen, many of the checked conditions are analogous to the example in Listing 4. The parameters passed through the function head represent the virtual machine to be migrated and the corresponding actor, with conditions in lines 3 and 4 ensuring that this actor is the defender and that *TARGETVM* can in fact be migrated. Line 5 serves to resolve *TARGETVM*'s current hypervisor, *OLDXEN*. However, this defense function also uses conditions that have not been presented yet and should not go unnoticed. Line 6 employs the *findAllWhere*-functor that is used to aggregate lists of elements that meet the conditions defined in the parentheses following that term. In this case, all elements *X* shall be collected, where *X* is of type *xen* and the VM to be migrated is not already in the list of *vms* of said *X*. The resulting list is assigned to variable *MIGRATIONPOOL*. Please note that this is different from using the *isInList*-functor as this would only check on single instances to be present in an attribute list, but would not return a list instance. Line 7 presents another functor, *pickFromList*, serving to randomly choose one element of the aggregated list referred to as *MIGRATIONPOOL* and store it as *NEWXEN*. While the following two conditions retrieve further information on the destination's subnet, that is the part of the network that the hypervisor is located in, lines 11 to 13 specify the target state, this time comprising *add* and *del* instructions to detach the target VM from the old hypervisor and attach it to the new one, while setting the corresponding subnet to that of the target hypervisor. Again, interested readers are referred

to the full scenario definition in Online Resource 1 that also contains attacks, defenses, and legitimate actions.

2.5 Attack simulation

Attack simulation is performed in a time-discrete round-based fashion with every attack, defense or legitimate action taking a specified number of rounds (i.e., its duration) and succeeding with a certain probability. A list of ongoing actions is maintained for each actor, keeping track of which actions have been started when, and in which round they are supposed to be executed. In each round, due defenses are performed before due attacker actions. Since the system state might have changed between starting and execution round of an action, the simulation engine first checks whether a due action's requirements are still fulfilled. Furthermore, for probabilistic actions a dice roll is used to decide on their success before engaging in any state modification. Once all due actions have been worked off, the defender(s) can initiate new actions. For this, they check if, according to their configuration, a defense action should be started and its requirements are fulfilled. If so, it is added to the list of ongoing defense actions with the respective execution time. Afterwards, the attacker(s) can initiate new actions according to configuration and the current system state, which are then added to the list of ongoing attack actions. For any type of action to be enlisted, Prolog verifies that related requirements are met.

The simulator employs a greedy attacker who attempts all available attack actions. Hence, as soon as an action's requirements are met, the attacker will start it. But once it has been started, the same action cannot be initiated with the same parameters again, as long as it is in the list of ongoing actions. For example, if the attacker started a phishing attack against a specific target, he has to wait for it to either succeed or fail, before launching the same attack against the same target again. Yet, if learning about a new target, the attacker may start a phishing attack immediately. The simulation engine does report costs for attempting or performing such attack actions but only counts the overall time till compromise, similar to the method used by P²CySeMoL [23] and pwnPr3d [25]. The alternative would be to limit the number of actions an attacker can execute in parallel and assign costs to each one of them. However, this would require an intelligent attacker with a strategy striving to make optimal decisions. These decisions, on the other hand, would need to incorporate observable system behavior while still being subject to imperfect knowledge. Considering this, the attacker himself would become an influencing factor so that evaluation results would not only reflect effects of the defense under test but also changes in attacker behavior, thus impeding fair comparison. This work, though, specifically focuses on investigating defense effects. In this regard, a greedy attacker is suitable for our purpose since all employed defenses "face" the same

potent attacker, making comparison ultimately fair. In the future work section we discuss how this framework might be extended to also consider intelligent attackers.

2.6 Metrics

For a fair comparison of defense techniques, it is important to have meaningful and independent metrics. These are based on aforementioned attacker revenue representing successful compromise that is logged in each round, with $r_j[i]$ denoting the revenue accumulated by the attacker up to round i for simulation j . To capture results, we define the following metrics: $rmax$ is the maximum accumulated revenue among all l independent simulations that are conducted for a given network and defense, $ravg_j$ is the average accumulated revenue per round for simulation j , and $ravg^m$ the median of these averages among all independent simulations. These are formally defined as:

$$rmax_j = r_j[n] \quad (1)$$

$$rmax = \max_{j \leq l} \{rmax_j\} \quad (2)$$

$$ravg_j = \sum_{i \leq n} (r_j[i]) / n \quad (3)$$

$$ravg^m = \text{median}_{(j \leq l)} \{ravg_j\} \quad (4)$$

The reason to conduct multiple independent simulations per benchmark network and defense is that the different actions exhibit different success probabilities. Consequently, development of attacker revenue may vary across simulations, even when repeated for the same benchmark network and defense. Considering this, l should be chosen high enough so that values for $rmax$ and $ravg^m$ are reproducible when performing another l independent simulations. Likewise, n , the number of rounds per simulation, should be chosen high enough so that the attacker has a chance to reach the maximum amount of achievable revenue and is not fundamentally limited by time. Otherwise, deviations in $rmax$ and $ravg^m$ cannot reliably be attributed to the tested defenses' effects but may be subject to time constraints. If this is given, $rmax$ shows how far an attacker can possibly infiltrate a network, so that deviations in this metric across defenses indicate if a defense was able to cut off attack avenues — or maybe opened new ones.

Yet, only looking at $rmax$ does not capture cases where a defense cannot prevent an attack, but slows it down. Many proposed MTD techniques do not claim to fully prevent attacks, but making them more costly in terms of required effort and time [2,3,5]. This is what $ravg_j$ is for. Relying on the average of accumulated revenue per round instead of gained revenue per round, $ravg_j$ accounts for both earliness and height of obtained revenue, yielding lower values

for cases where revenue acquisition is delayed, even though $rmax$ may be the same. This way, the two metrics complement each other. Deriving $ravg^m$ from all $ravg_j$ makes this metric resistant to outliers and condenses results from l independent simulations per network and defense into a single value.

However, while these metrics already aggregate results, they are only concerned with one particular benchmark network at a time. When dealing with hundreds of networks, as may easily be generated and tested using the suggested approach, $rmax$ and $ravg^m$ on a per-network basis are not as helpful anymore. To make sense of this information across a multitude of benchmark networks, the impact of single defenses on $rmax$ and $ravg^m$ needs to be classified. To accomplish this, we first take a look at $rmax$ which represents the worst case for a given network and defense. For each benchmark network, we compare values of $rmax$ for all active defenses with that of *no defense*, to determine if these are smaller, equal, or larger and categorize them accordingly. Applying the same scheme to $ravg^m$ is not possible, though, since $ravg^m$ is derived from average values of all independent simulations, so that it is hardly ever the same across different defenses, even though values may be similar. This makes it hard to decide when values are genuinely different or roughly the same. Therefore, we use $ravg^m$ together with its 95% confidence interval. This is calculated using the approach of McGill *et al.* [31] that relies on a Gaussian-based asymptotic approximation of the standard deviation and is defined as:

$$M \pm 1.7 \frac{1.25R}{1.35\sqrt{(N)}} \quad (5)$$

In this expression, M represents the observed median, R is the interquartile range defined through $R = Q3 - Q1$ and N is the number of observations which we denoted as l . For each defense we then check if the confidence interval lies entirely above or below the confidence interval of *no defense*. If so, we classify its $ravg^m$ as “larger”, respectively “smaller”. This is in line with McGill *et al.* [31] who state that non-overlapping confidence intervals indicate significant difference. If intervals overlap, values are generally classified as “equal”. How these metrics are used in practice to analyze real data, is shown in Sect. 4, where experimental results of the following case study are presented.

3 Case study

For our case study we assume a mid-sized corporate network consisting of nine subnets. These allow for a considerably more complex network hierarchy than the three subnets used in previous investigations [8], let alone scenarios that consist

of merely three hypervisors and five unspecific VMs [24]. While this generally requires more lateral movement for the attacker to compromise all valuable resources, it also provides more occasions for defenses to unfold their potential and learn about their effects. However, the most important difference is that in this case study we employ a fuzzing approach. That is, network instances are not modeled manually as before, but are derived from a single high-detail scenario definition that serves as a template for automated diversification. To test this functionality and investigate the spectrum and quality of results obtained from such auto-generated network instances, we opt for a larger experiment. Instead of evaluating defenses in only two benchmark networks, we scale up investigation by factor 100 and have our framework automatically generate 200 instances to perform simulation on. These 200 instances share the same basic network layout and general connectivity, but differ in a number of aspects which have been declared subject to probability and conditions. In particular these include:

- No. of clients and servers in different subnets
- Operating systems of clients and servers
- Server types (e.g. CRM, SQL, Webserver etc.)
- No. of Xen hosts within subnets to migrate VMs
- Existence and distribution of different vulnerabilities in:
 - Hardware
 - OS
 - Applications
- Reuse of credentials
- Storing and caching of credentials
- Firewall misconfiguration

However, respective characteristics are not chosen fully randomly, but are determined under consideration of other probabilistically chosen attributes so far. This means that if diversification lead to the instantiation of a high number of servers, the scenario definition will scale up the range of potentially chosen server applications to deploy. These, in turn, impact the range of potentially instantiated client applications and inserted firewall rules, to give just one example. Similarly, instantiation of additional SQL servers or re-use of existing ones is only considered, if there are applications that need those in the first place. This is to ensure that created benchmark networks are plausible and consistent, allowing us to evaluate attack and defense performance in different, yet equally realistic scenarios and get a better understanding of their impact on security. In the following, the used defenses, network layout, software landscape and revenue distribution, as well as vulnerabilities and attacker actions will be explained.

3.1 Defense techniques

We employ four defense techniques to investigate their impact on attacker progress: *IP shuffling*, *VM live migration*, *VM cold migration*, and *VM resetting*. The first three have been chosen for their frequent occurrence in MTD literature and the existence of prototypical implementations. *VM resetting*, was included for being integral to *cold migration* and having a potential impact on security itself. In addition, we also run simulations with *no defense* to put performance of the different defenses into perspective.

IP shuffling is one of the most frequently suggested Moving Target Defenses [11,20,26,28,33], advocating the repeated change of IP addresses of communicating entities. The general concept of changing network addresses as a means of defense is often referred to as network address space randomization (NASR) and may also comprise ports, and in rare cases even MAC addresses [29]. The intention is to impede the attacker through invalidating previously acquired knowledge of addresses and making guessing significantly harder by leveraging large address spaces. The implementation we use throughout our case study only considers IP addresses and is therefore denoted as *IP shuffling*.

VM live and cold migration propose to repeatedly relocate VMs across various physical hosts. Doing so for the purpose of defense has been addressed in previous work dealing with MTD and network defense in general [1,6,17,22,24,34,39], with suggested schemes generally differing in whether a VM's state is preserved or not. State preserving migration that we denote as *live migration* intends to seamlessly move a VM out of the attacker's reach while keeping it as is. Attacks that *live migration* is supposed to fend off are such that rely on certain connectivity [24] or co-location of VMs [22,39]. Another form of VM migration we refer to as *cold migration* suggests to not only migrate VMs but reset them to a default state, reverting any potential modifications caused by the attacker [3,18]. Additionally, such VMs (may) receive new IP addresses (e.g. through DHCP), thus implementing a form of *IP shuffling* whenever migrated. In the scope of this work, *cold migration* combines moving and resetting of VMs while providing them with new IP addresses.

VM resetting suggests the sole recurring reset of virtual machines. Prior to the advent of MTD, approaches such as SCIT [10] already promoted the idea of resetting VMs to previous, presumably secure states to prevent attackers from reaching persistence. Though not being a Moving Target Defense in the classical sense, it has been included to determine its individual impact on security. This allows us to dissect performance measurements of *VM cold migration* and relate its results to those of the defenses it is composed of.

A summary of employed techniques can be found in Table 1. By using the same defenses as in our earlier

Table 1 List of defenses that are employed in the case study for comparison of performance, as opposed to using *no defense*

Name	Description	Impact
Live migration	The VM is migrated from one physical host to another without loosing its state	Moving a VM changes the physical connectivity and hence communications paths
VM resetting	The VM is restarted from read-only memory, loosing all state information	Any remote code execution privileges on the VM previously gained by the attacker are removed
IP shuffling	A new IP address is assigned to the VM	Knowledge of the IP address previously gained by the attacker is removed
Cold migration	The VM is migrated to another physical host, restarted from read-only memory, and assigned a new IP	This is the combination of <i>live migration</i> , <i>VM resetting</i> , and <i>IP shuffling</i>

experiment, we get the chance to investigate if the previously provoked incidental security degradation caused by individual defenses can also be observed when employing automated and unbiased diversification. Should this be the case, benchmark fuzzing would not only be convenient but also suitable to produce scenarios that yield meaningful insights. Furthermore, the increased scale, complexity, and diversity may reveal additional effects and shed light on how these are distributed.

Note that the real-world implementation and application of such defenses may come with challenges. For example, the blunt shuffling of IP addresses might not only impede attackers but also break legitimate communication if no precautions are taken. To account for this, in our simulation, legitimate communication relies on using DNS names. Likewise, the resetting of VMs renders any service useless that stores data locally, which is why we exempt VMs from *cold migration* and *VM resetting* that host databases or serve as storage.

Considering the introduction to modeling attacks and defenses from Sect. 2.4, the framework is not limited to these four defenses but can easily incorporate other ones. Depending on their mechanism, function definitions may even be similar. Service migration [1], for example, an MTD technique advocating to migrate services from high-risk VMs, could be implemented in nearly the same way as VM migration, that is detaching the target application from its current OS and assigning it to another one while adapting related attributes. However, the framework can also model static defense mechanisms that do not change the system state proactively but affect the evaluation of other actions. Firewalls are a suitable example for such defense techniques. While not being defined as a function that the defender calls, they are implemented as additional conditions that decide whether or not communication is possible in the first place by checking for applicable firewall rules. In turn, defenses

Fig. 2 Fixed subnets, represented as circles, that are randomly populated and interconnect through firewalls. Communication is possible as indicated by the different arrow types

such as anti virus programs would check for the existence of code signatures to detect exploits, thus inhibiting related attack actions.

3.2 Network layout

The scenario definition employs a high degree of freedom to generate diverse benchmark networks in large quantities. Yet, the general network structure is fixed, consisting of nine specific subnets. That means, irrespective of randomized properties such as the number of hosts in the demilitarized zone (DMZ), their configuration, or installed applications, there will be a DMZ in every benchmark network. The same goes for other subnets and structural properties so that all instances share a general layout as shown in Fig. 2. The depicted circles represent the subnets that form the network

which is connected to the internet, the starting point of the attacker. The subnets “DMZ” and “server” are composed of physical hosts with hypervisors and virtual machines on top to provide services for regular operation. The “DMZ” is exposed to the internet, whereas the “server” subnet can only be reached from the network’s inside. “Accounting” and “marketing” depict the low-privileged clients in this setup, whereas the “admin” subnet contains clients with extended privileges and connectivity within the network. The subnets “engineering client” and “engineering server” comprise servers and clients with limited connectivity to the rest of the network, hosting applications and performing tasks to sustain operation of industrial machines located in the “manufacturing” subnet.

In Fig. 2, all subnets are connected through different types of arrows, representing different types of possible communication. This is either IP and port-based, out of band via removable media, or through e-mails. However, apart from requiring general connectivity, IP and port-based communication is subject to firewalling. In the given network model, all subnets are separated through firewalls that prevent any IP and port-based communication from one subnet to the other, unless there is a firewall rule that explicitly allows communication for two entities based on their IP addresses and the ports they use. Furthermore, communication within the “DMZ”, “server”, and “engineering server” subnets is filtered so that the different Xen servers and their guest OSes cannot communicate unimpeded. Communication within client subnets is not firewalled, though.

In our case study, firewall rules to enable communication are deployed on the basis of instantiated applications, as well as accidental misconfiguration. The former represent legitimate cases in which the scenario definition determined specific clients to communicate with corresponding servers and consequently instantiates such rules. The latter is to represent cases in which rules have been inserted incorrectly or simply been forgotten, which may plausibly occur in large networks, thus enabling unintended communication paths. Yet, misconfiguration is only employed on a small scale. In this context, it should be noted that firewalls are defenses themselves. However, the scope of this work is on proactive defenses. Furthermore, we perceive firewalls to be fundamental for any network, which is why they are not subject to analysis in our case study but employed by default.

The depicted hierarchy has been chosen to account for company networks with subnets for development and manufacturing that, to a certain degree, need to be isolated from regular network traffic for their criticality to operations or confidentiality reasons. As can be seen in Fig. 2, network-based communication to “engineering and manufacturing” is restricted to the server subnet that may be required for keeping numbers of produced parts in sync with accounting, for example. However, there might also be unofficial

communication from and to engineering clients via removable media to represent users that try to bypass inconvenient restrictions. This property is inserted probabilistically for individual clients of the respective subnets. Communication is less restricted in the part of the network that can be considered as the office IT. There, clients access application servers in the DMZ and server subnet, while admins have to manage these. Exposed servers from the DMZ need to communicate with the server subnet for authentication services and the like. Additionally, clients from marketing and accounting, as well as admins are able to receive e-mails.

Ranges of the numbers of physical nodes, VMs, as well as applications in our scenario definition are depicted in Table 2. As one can see, for applications in some client subnets, these numbers may rise quickly. This is due to the fact that the scenario definition automatically scales different aspects of the generated benchmark networks to keep them plausible and realistic. For this purpose conditions are employed to check on the specific shape of previously determined probabilistic characteristics, influencing how probabilistic scenario generation proceeds. Should the number of physical machines with hypervisors in the server subnet be high, for example, the definition selects the number of virtual machines to be deployed from a higher range of potential values, instead of a lower. While specific numbers are still subject to fuzzing, this mechanism ensures that 10 physical hosts with hypervisors are not encompassed with merely two VMs but rather 20. This, in turn, triggers the number of server applications to be selected from a higher range with specific instances chosen from a list of application candidates, increasing possibilities to interact. Otherwise, the high number of VMs would simply bloat the model without serving any purpose. Eventually, a higher number of server applications causes numbers of clients and client applications to be scaled up. In this process, specific clients are randomly chosen to be legitimate peers of selected server applications and equipped with corresponding client applications. As a result, legitimate interaction with the server applications is not limited to only few clients that happen to communicate with every server, but spread across numerous clients that may yield different information and capabilities when being compromised. At the same time, this pairing of client and server applications triggers the instantiation of firewall rules to allow for communication between respective addresses and ports. This logic has been used throughout the definition file to enable the generation of versatile and realistic benchmark networks.

3.3 Software landscape and revenue distribution

The applications that are instantiated depending on probability and conditions have been selected with regard to functionality that is regularly required in a corporate context. These comprise Active Directory (AD) Servers, Microsoft

Table 2 Range of possible scenario size

Subnet	No. of Phys. Nodes	No. of VMs	No. of Apps
Remote	1–3	0	6–45
Internet	2	0	6–12
dmz	2–3	4–12	5–14
Server	4–9	11–24	15–32
Printer	1–3	0	0
Engineering clients	6–9	0	3–51
Engineering server	2–5	5–12	6–15
Manufacturing	1–4	0	1–4
Marketing	4–7	0	24–98
Accounting	4–7	0	24–105
Admin	2–4	0	2–4
Total	29–56	20–48	92–380

Table 3 Potential applications per subnet

Subnet	Potential server applications
dmz	Apache2, Tomcat, SQL, Exchange, Remote Shell/Desktop
Server	Active Directory (AD), SQL, SAP, Max DB, Veritas Netbackup, Tomcat, Marketing (generic), Datev, SMB, Remote Shell/Desktop
Engineering server	Git, Licensing (generic), SQL, SAP, Max DB, Remote Shell/Desktop
Manufacturing	<i>generic application for deployment</i>
	Potential client applications
Internet	DMZ: Apache2, Tomcat, Exchange
Remote	DMZ: Apache2, Tomcat, Exchange; Server: AD
Marketing	DMZ: Apache2, Tomcat, Exchange; Server: AD, Veritas Backup, Tomcat, Marketing
Accounting	DMZ: Apache2, Tomcat, Exchange; Server: AD, Veritas Backup, Tomcat, Datev, SAP
Admin	DMZ: Apache2, Tomcat, Exchange, SQL, Remote Shell/Desktop; Server: AD, Veritas Backup, Tomcat, Datev, SAP, SQL, Max DB, Remote Shell/Desktop

Exchange, Apache Tomcat, SAP for manufacturing and accounting, but also different types of database servers, to name just a few. Applications come with typical actions one might reasonably expect to exist as to allow for interaction. In that sense, the Apache2 server allows to retrieve websites, whereas the OSes allow to read from the file system, for example. On occasions where there is no obvious real world application to represent a certain functionality, generic applications have been modeled and equipped with plausible actions. An example for such a generic application is the marketing server that is assumed to be accessed by client applications on nodes in the marketing subnet and allows, depending on the given privilege, to retrieve certain types of information. Table 3 gives an overview of the different applications that the benchmark network generator chooses from, in order to equip server and client nodes in the respective subnets. In our case study, more than one instance of an application can be deployed per subnet. This is not only true for SQL servers that are frequently used exclusively for

specific services, but also for other applications that may be used on various occasions. For each application to be created, the benchmark network generator re-evaluates possible configuration parameters, as well as potentially existing vulnerabilities so that the resulting application instance exhibits a different set of characteristics than previous ones. Using the example of a Tomcat server, these parameters range from whether or not a local, a remote, or no database at all is being used, which affects connectivity and may even introduce new VMs in the respective subnet. Yet another parameter is the activation of the JMX port for maintenance, which may pose a threat if not secured with credentials, as is the default. The list of client applications contains additional information about the target subnet.

Just like applications, operating systems are subject to automated diversification. In this experiment we differentiate between Windows, Linux and Xen. While clients in “marketing” and “accounting” always run Windows, the hypervisors in the “DMZ” and “server” subnet run Xen. Yet, “admin”

clients and the VMs on top of the hypervisors are randomly equipped with either Windows or Linux, unless the application they host (e.g. Exchange or Active Directory) explicitly requires the one or the other. These operating systems, in turn, differ with regard to OS-specific parameters and vulnerabilities whose exact shapes are equally subject to fuzzing and determined on a per-host basis. That means, for every operating system to be created, the list of applicable parameters such as the activation of RDP (remote desktop protocol) on Windows, SSH (secure shell) on Linux and Xen, or the SMB (server message block) protocol is randomly composed. The same goes for OS-specific vulnerabilities such as SMB-related *EternalBlue* or privilege escalations on the Linux shell, to name just two examples.

For every server application, at least one element of type *usefulData* is instantiated, holding the revenue that attackers may acquire. These are directly assigned to said applications, assuming that valuable data are located on servers. Given that the total number of server applications is probabilistic, so is the number of *usefulData* elements. In addition, the number of *usefulData* elements is subject to further diversification based on the probabilistic number of clients using an application. Mailboxes, being one example of *usefulData*, are only generated if there is a mail server. Their number, however, depends on the number of clients. As a result, the total amount of available revenue varies across benchmark networks, affecting the amount an attacker may gain. Consequently, comparing absolute numbers of gained revenue across benchmark networks is not advisable. However, this does not affect comparing gained revenue across different defenses within the same benchmark network, which is exactly what the previously introduced metrics postulate and will be demonstrated in Sect. 4. Should a hypothetical experiment require the amount of revenue to be fixed across all benchmark networks or correlated to the number of existing hosts, for example, the scenario definition must be designed so that the number *usefulData* elements is either fixed or directly linked to the instantiation of nodes. In this regard, the experimenter is free to choose if and how to relate available revenue to the shape of benchmark network instances.

3.4 Exploits and legitimate actions

As outlined in Sect. 2.4, the attacker's interaction with the system is realized through functions representing legitimate actions and exploits, both of which come with individual requirements and effects. In that sense, querying SQL databases may require knowledge of credentials, for example, and, if successful, lead to the attacker learning stored information. Performing an *EternalBlue* exploit, though, requires the target node to run a vulnerable version of the SMB service and will grant remote code execution (RCE) privileges if successful.

Detailed protocols of successful attacks like that of Phineas Fisher on Hacking Team¹ have revealed that for attackers to reach their goals, legitimate actions are as relevant as vulnerabilities and their exploits. Consequently, legitimate actions are integral to our model. These have been determined by scanning through the list of applications and operating systems employed in our model and identifying their core functionalities, as well as those that are relevant to general network communication. To name just a few, these actions comprise fetching e-mails, pinging hosts, opening remote shells, and reading from filesystems, all of which have been implemented using our modeling language. Related vulnerabilities have been obtained from the CVSS database and metasploit, filtered with regard to age and score as to only consider those with a score higher than eight from the years 2016 to 2019. Vulnerabilities from this list have been incorporated into the model if exploiting them yielded any of the following effects:

- allow restricted or privileged read access to data
- grant restricted or privileged remote code execution
- escalate privileges

Corresponding exploits have been modeled just like legitimate actions, requiring different conditions to be met, including the presence of respective vulnerabilities whose distribution among hosts is subject to probability. In consequence, Windows instances vulnerable to *EternalBlue*, Tomcat installations with their JMX interface activated, and other potentially vulnerable configurations vary across benchmark networks.

When comparing the different legitimate actions, but also exploits, similarities in underlying mechanisms, as well as effects become apparent. Actions such as “SQL query” and “fetch e-mail”, for example, can be classified as authenticated reading of network-based resources. Many exploits, on the other hand, provide attackers with RCE privileges requiring no authentication at all. Based on these reoccurring patterns, generic actions and exploits have been crafted that are used to equip generic applications such as the marketing server with plausible means of interaction.

All in all, this resulted in 48 actions an attacker can directly call, with Table 4 giving an overview of how these have been classified along different dimensions. While *type* differentiates between exploits and legitimate actions, *execution environment* distinguishes the setting in which these may be launched. In our case study, this may either be through network connectivity to the target, RCE privileges on the target system, RCE privileges on any system that resides on the same physical host as the target, or through other

¹ A detailed outline of how the attacker proceeded is provided at <https://www.exploit-db.com/exploits/41915>.

Table 4 Attacker functions (48) grouped by specific attributes

Dimension	Shape	Number
Type	legitimate	24
	exploit	24
Execution Environment	Network	37
	Local System	4
	Local Host	4
	Others	3
Execution Restriction	Authenticated	21
	Unauthenticated	27
Effect	Read Assets from Application (full/restricted)	5/9
	Read all data from OS	7
	Read all data related to Application	3
	Read only IP addresses	2
	Remote Control (system/user)	20/2

channels such as e-mail or removable media. *Execution restriction* indicates whether the attacker needs to authenticate with credentials or not and, finally, *effect* categorizes actions with regard to the changes they cause to the system state. “Reading assets from applications” refers to obtaining data that have a defined value that we use to measure attacker revenue. This may be credit card information stored in a database that the attacker can get hold of by different means. Differentiating between “full” or “restricted” simply refers to the privilege level. “Read only IP addresses” results in the attacker learning IP addresses. These have no inherent value but are needed for further attack steps. “Reading all data” from either applications or complete operating systems comprises extraction of any information type, including IP addresses, DNS names, usernames, passwords, etc. as well as aforementioned assets. This may be the case when getting access to the filesystem to retrieve configuration files of different services. Lastly, “remote control” subsumes all actions that grant remote code execution, be it through legitimate ways such as a RDP or exploiting vulnerabilities. An exhaustive list of all implemented actions can be found in Online Resource 1, comprising definitions of their requirements and effects.

Apart from requirements and effects, each action has two additional properties that have already been introduced in Sect. 2. These are *duration* and *success probability*. Although CVSS database entries include information that may indicate these values for a given exploit, specific figures cannot be derived from a given score. Therefore, duration and success rate have been manually determined on the basis of a vulnerability’s description, the underlying mechanism, and the availability of exploit code (e.g. in metasploit). It should be noted, however, that these values could potentially be optimized with additional data from real world attacks. The success probability of legitimate actions was generally con-

sidered to be high and durations have been defined in relation to other actions.

4 Experimental results

For every combination of the 200 benchmark networks and five defense configurations, we performed $l = 100$ independent simulations with $n = 12000$ rounds each. The number of rounds n was chosen high so that for simulations with *no defense* in any given network, the attacker could, and in 91.8% of the cases did, reach the maximum revenue that was achievable in the presence of *no defense*. Using the metrics r_{max} and r_{avg}^m as presented in Sect. 2.6, the results of two specific benchmark networks are presented in detail. These have been chosen for being representative cases of dominant effects that can repeatedly be observed across benchmark networks, as Sect. 4.3 will outline in more detail. Furthermore, they serve as interesting examples for yielding contradicting verdicts with regard to the utility of certain defenses, making them adequate candidates to motivate the critical discussion of MTD techniques. However, other benchmark networks would have been equally eligible for that purpose. Afterwards, aggregated results considering all investigated benchmark networks will be presented according to the classification scheme that has been introduced together with r_{max} and r_{avg}^m .

4.1 Results of individual benchmark networks

Figure 3 depicts “notched” boxplots representing the average accumulated revenue per round r_{avg_j} , as well as the maximum attacker revenue r_{max_j} for networks 96 and 157

Fig. 3 Boxplots representing attacker performance for scenarios 96 and 157 based on (i) the average accumulated revenue per round avg_j across different defenses and 100 independent simulations each and (ii) the maximum accumulated revenue $rmax$ at the end of each simulation

Fig. 4 Histogram of the attack simulations over 200 networks with **a** showing how often each defense resulted in a larger, the same, or smaller $rmax$ compared to *no defense*. In **b** the number of networks is shown in which avg^m for a chosen defense is significantly smaller or larger than that of *no defense*

based on $l = 100$ simulations per defense.² In each plot, the upper and lower boxes depict the top and bottom quartiles, representing the 25% of values above and below the median. The horizontal line that separates top and bottom quartile along the notch, is the median. For boxplots that display the average accumulated revenue per round, this is avg^m . Whiskers, where present, indicate variability of values outside the quartiles, while outliers are represented as crosses. The notches in the boxplots represent the 95% confidence interval for the median. Simulations with *no defense* were performed three times, yielding two control groups. Since these are based on the same inputs as the reference case, they are subject to the same probability distribution. Consequently, if sample size and simulation duration have been chosen adequately to employ said metrics, there should be no significant difference. Fig. 3a, b show boxplots of avg_j and $rmax$ for network 96, allowing for interesting observations. Most notably, values for $rmax$ are considerably higher for both types of VM migration as opposed to employing *no defense*. This means, that these defenses did not only fail to prevent attacks, but opened up new attack opportunities. The

qualitative analysis of corresponding simulation logs reveals that this results from changed connectivity and co-location of VMs that occasionally increases the number of viable attack paths. The other defenses yield the same $rmax$ as *no defense*, meaning that the number of successful attacks is the same. Looking at the average accumulated revenue we can see that *cold migration* and *live migration* also have a negative impact on this performance indicator, yielding higher values than *no defense*. *IP shuffling* has no considerable impact while *VM resetting* was able to slow down the attacker. Note that *cold migration* exhibits a lower average accumulated revenue per round than *live migration* which is explained by the fact that *cold migration* also incorporates *VM resetting*. What this tells about benchmark network 96 is that its initial configuration, that is, before moving any VMs and changing corresponding firewall rules, was comparatively secure as opposed to constellations in which VM positioning differs. While the attacker was able to penetrate the network and extract some of the assets that yielded revenue, reaching and compromising all of them was simply not possible. This dead end opened up when VM migration was triggered, creating the possibility that other vulnerable VMs move into the attacker's reach, enabling further lateral movement.

² Online resource 2 provides boxplots of all networks, including data and Python scripts to generate them.

In contrast, network 157 paints a different picture. As shown in Fig. 3d, in this network, $rmax$ is the same across all defenses, meaning that none of them decreased or increased the number of successful attacks. However, as shown in Fig. 3c, both types of VM migration, as well as *VM resetting* have a throttling effect on the attacker. Furthermore, one can see that *cold migration*, being a combination of the other defenses, has the strongest impact. Judging from this case, VM migration appears to be a capable defense. As the qualitative analysis of simulation logs reveals, benchmark network 157 exhibits a configuration where exploitable vulnerabilities are “lined up” in a way that enables the attacker to laterally move throughout the network and compromise resources. The moment that VM migration starts to change this configuration, the attacker cannot proceed unimpeded anymore but is considerably slowed down as reconnaissance needs to be repeated to determine if and when a vulnerable machine is within reach again. In that sense, network 157 exhibits a comparatively vulnerable default configuration that could be improved through changing VM placement.

4.2 Comparing defenses among 200 benchmark networks

The important question arising from these findings is which types of effects exist for the different defenses and how they are distributed. So far, we have had a look at two exemplary network instances and observed that their results differ. In this section, we compare defenses among all 200 networks that have been generated with our benchmark fuzzing approach. To accomplish this, the classification scheme as presented in Sect. 2.6 is used to determine whether the different defenses impacted $rmax$ and $ravg^m$ to measurably deviate from employing *no defense*. Doing so across all 200 benchmark networks and counting occurrences allows for a comprehensive overview. Figure 4a shows respective histograms of all defenses together with the two control groups (where *no defense* was active) across all 200 scenarios. One can see that for *IP shuffling* and *VM resetting*, as well as the two control groups, $rmax$ is never higher than that of *no defense*, thus never causing any security degradation. On the other hand, in the presence of *cold migration* and *live migration* in 69, respectively 74, out of the 200 networks, $rmax$ increases. That means, in more than a third of all cases, migration actually has a negative impact on security. A reduction of maximum attacker revenue is only observed in very few cases (i.e., *cold migration* 6/200, *IP shuffling* 5/200, *live migration* 0/200 and *VM resetting* 9/200). In this regard, the considered defenses hardly ever keep the attacker from reaching his goal as compared to *no defense*. What should be noted is that *VM resetting* and *IP shuffling* appear to have hardly any impact on $rmax$ at all. This may imply that these techniques simply do not affect security. However, it may also indicate

that $rmax$ alone is not sufficient to capture their effects. This becomes apparent when looking at their impact on average accumulated revenue, which is significant.

Figure 4b depicts the histogram of the average accumulated revenue per round. Again, including results from the two control groups for which simulations with *no defense* have been repeated. Note that a significance level of 95% means there is a probability of up to 5% that a median is falsely classified as different. In our control groups we observe deviation in 3% of the cases so that our assumption of a classification error smaller than or equal to 5% is not violated and results are generally reproducible. As can be seen, *cold migration* and *live migration* cause an increase of average accumulated revenue in 30.5%, respectively 43% of the cases, while *VM resetting* and *IP shuffling* exhibit no such effect. On the other hand, *live migration* reduces the average accumulated revenue in only 13.5% of networks which is the lowest followed by *IP shuffling* (23%), *cold migration* (37.5%), and *VM resetting* (40.5%).

In conclusion, one can state that *cold migration* may have both positive and negative effects. However, in our benchmark networks, its positive impact on security is lower than that of using sole *VM resetting*. *Live migration* affects security mostly negatively. Consequently, both types of VM migration are not advisable since they open attack paths that would otherwise not exist. Instead, *VM resetting* and *IP shuffling* have the largest positive impact on security without any downsides. Among the two, *VM resetting* increases the security level considerably more (40.5%) than *IP shuffling* (23%). Note, that these findings concern the defenses’ general effectiveness, given their duration times as delimiting factors. To what extent the most effective defense is attractive for a defender to employ, however, may depend on factors such as strategy, environmental conditions, and the cost that result from these. Nevertheless, in this specific scenario of a mid-sized enterprise network with given vulnerabilities and exploits, *VM resetting* performs the best.

4.3 Coverage of benchmark fuzzing

The overarching question, whether our fuzzing approach is able to yield equally meaningful results as the manually tweaked benchmark networks, can be clearly answered with yes. Not only did fuzzing produce the same cases as seen in previous work, it also revealed new cases that had not been observed yet.

Across the 200 benchmark networks, 37 unique combinations of defense effects could be observed, with every combination being characterized by the security impact each of the four defenses had on $rmax$ and $ravg^m$. As outlined before, impact was measured in comparison to simulations where *no defense* was employed and could either have an increasing, decreasing, or no effect at all. Among these com-

binations, some are more dominant than others, so that the ten most frequently observed combinations already account for 161 of the 200 networks. These effect combinations are summarized in Table 5. The descriptions of the effect combinations only specify the impact of defenses that measurably deviate from employing *no defense*. Obviously, *E1* is an exception to this rule as in this case none of the defenses exert any measurable effects. To give an impression of the proportions of these effects, Fig. 5 presents a pie chart, showing how effect combinations are distributed among the 200 benchmark networks. Note that the share marked as “rest” subsumes the 39 remaining benchmark networks that exhibit one of the other 27 observed effect combinations which occurred less frequently. The two most dominant effect combinations that measurably deviate from employing *no defense* are *E2* and *E3* that have been discussed at the examples of scenarios 96 and 157 where migration either degraded or improved security. However, extending simulation to include more networks of higher diversity also reveals that *IP shuffling* was able to slow down the attacker in a considerable number of cases, as indicated by *E4* and *E10*. While not as effective as *VM resetting*, it still increases security. Furthermore, cases are revealed where *cold migration* and *live migration* have opposing effects on avg^m (*E7*) that result from the *VM resetting* property of *cold migration*, for example. This highlights that relying on a few hand-crafted benchmark networks is not enough to analyze defense techniques. Instead, it is advisable to base analysis on many benchmark networks that, using fuzzing, can be derived automatically from one single scenario definition. A condensed table of all effect combinations can be found in Appendix B.

5 Performance and scalability

Performance and Scalability of the presented approach are relevant in two regards. Firstly, the automated generation

Fig. 5 Shares of dominant effect combinations, figures in absolute quantities

and fuzzing of benchmark networks, and secondly, simulation that is conducted based on the generated networks. In the following, we present measured timings for both these steps based on the 200 benchmark networks covered in the previously presented case study, as well as 200 additional benchmark networks derived from a newly created scenario definition. This second scenario definition makes equal use of fuzzing to generate diverse benchmark networks. Yet, it has been composed so that resulting network instances are smaller with regard to the number of nodes and applications that are distributed across only three subnets “DMZ”, “server” and “client”. Resulting networks resemble the manually crafted networks that have been employed in our previous work and serve to broaden the spectrum of networks to evaluate the framework’s performance and scalability.

Computation was conducted on a 16-core Ryzen (3950x) CPU from AMD. Yet, all presented timings reflect single-

Table 5 Dominant effect combinations across all networks

Id	Effects	Combined impact
E1	No defense has any impact on $rmax$ and avg^m	Neutral
E2	<i>Cold</i> and <i>live migration</i> increase both $rmax$ and avg^m	Negative
E3	<i>Cold</i> and <i>live migration</i> , as well as <i>VM resetting</i> decrease avg^m	Positive
E4	<i>Cold migration</i> and <i>IP shuffling</i> reduce avg^m	Positive
E5	<i>Cold migration</i> and <i>VM resetting</i> reduce avg^m	Positive
E6	All defenses reduce avg^m	Positive
E7	<i>Cold migration</i> and <i>VM resetting</i> reduce avg^m , <i>live migration</i> increases avg^m	Mixed
E8	<i>Cold</i> and <i>live migration</i> increase both $rmax$ and avg^m , <i>VM resetting</i> reduces avg^m	Mixed
E9	<i>Live migration</i> increases $rmax$ and avg^m , <i>cold migration</i> only increases $rmax$	Negative
E10	<i>Cold</i> and <i>live migration</i> increase $rmax$ and avg^m , <i>IP shuffling</i> and <i>VM resetting</i> decrease avg^m	Mixed

threaded operation. Upon generating multiple benchmark networks or performing numerous independent simulations at once, the framework can parallelize these across multiple threads, thus cutting total time to a fraction, depending on the utilized CPU.

5.1 Benchmark network generation and fuzzing

To measure the performance of automated benchmark network generation and diversification, the time required for generating each instance has been recorded. For benchmark networks based on the original scenario definition (i.e. the one in the case study) this took, on average, 162.8 seconds per benchmark network. For the smaller networks derived from the additional scenario definition, the average generation time was 57.1 seconds. Timings of individual networks vary around these averages, depending on the size of the instance to be generated. This size, in turn, depends on the probabilistically determined number of nodes, applications and other elements to be instantiated. The processing of loops and if-statements causes no measurable load beyond what is needed to process the expressions they enclose for the given number of iterations. This means that creating 100 nodes with help of a loop causes the same computational load as unrolling the loop and having the generator create 100 individually defined nodes. Relating generation time to the size of a network in terms of the number of nodes, yields insights on how benchmark network generation scales. While other aspects such as the number of applications or elements of type *usefulData* still vary across benchmark networks featuring the same number of nodes, the realistic scaling as outlined in Sect. 3.2 ensures that their quantities are related, to a certain degree. Furthermore, the number of interconnected nodes is a comparatively intuitive measure to describe a network's size.

Figure 6 presents generation time as a function of benchmark network size for all 400 benchmark networks, with bullets/dashes depicting the mean generation time, and the vertical lines indicating the ranges of observed maximum and minimum time for networks of related size. As one can see, the generator's performance scales almost linearly for the considered range of benchmark network sizes, even across the two scenario definitions, with variations being related to the varying numbers of other elements in individual benchmark network instances.

5.2 Simulation

Analogous to benchmark network generation, simulation performance is measured in required time that is logged for every conducted simulation. Obviously, average reasoning time increases with network size as the number of potential targets rises, all of which need to be considered by the

Fig. 6 Time required to generate benchmark networks in dependence on the number of created nodes

Fig. 7 Simulation duration in dependence on the number of created nodes

simulator. However, simulation time also depends on the number of applied state changes that result from successful attack and defense. This is due to the fact that the simulation engine employs tabling, a feature of SWI-Prolog to cache and look up results of previous reasoning instead of recomputing them. This considerably reduces computational effort and time when determining an actor's options. Yet, the cache must be rebuilt whenever the system state changes as reasoning may come to other conclusions in the new setting. In consequence, a high number of successful interactions increases simulation time, whereas little interaction allows to use cached information for longer periods of time. Note that for simulations where defenses are active, every successful defender action causes tabling to be refreshed to check whether or not this affected the attacker's options.

Figure 7 visualizes simulation time in relation to benchmark network size for all 400 instances, again using bullets/dashes to represent means, and vertical bars for the ranges from minimum to maximum. These values are derived from all simulations for all defenses that have been performed for networks of given size. As can be seen, the time required for simulation is subject to variations with differences between upper and lower bound growing larger with benchmark network size. This is due to the simulation time's aforementioned dependency on both benchmark network size and number of successful interactions. On the one hand, larger networks may allow for more interaction than smaller networks, simply for the number of potential targets so that tabling must be refreshed more often. On the other hand, network size increases reasoning time on a per-action basis as the simulator has to consider more potential options. Consequently, for high-interaction benchmark networks, simulation time grows non-linearly with network size. The average, however, is lower than that since not all networks exhibit a shape where actors may exert state changes in each and every round. Please note that the semblance of simulation time decreasing beyond network sizes of 85 nodes is simply due to the fact that the investigated sample of 400 benchmark networks only features few networks of this size. As a result, the observed range of simulation time for such networks is limited.

Considering that defenses add to the degree of interaction by causing state changes themselves, but also triggering or enabling subsequent attacker actions, their impact on simulation time shall not go unnoticed. For this purpose, Table 6 presents the average simulation time as a function of employed defenses across all benchmark networks, together with the overall average to put them into perspective. As can be seen, employing *no defense* yields the best performance. Given the previous outline, this is not surprising as it does not include any defender actions. *Cold migration*, in turn, requires the most time which is due to its numerous effects on the system, that trigger the attacker to repeat reconnaissance and explore new potential attacks paths. Note that the comparatively long simulation time of IP shuffling is due to the fact that this defense can be used in short intervals in our simulation, thus increasing the interaction count.

6 Related work

A considerable amount of existing research addresses the challenges of evaluating MTD techniques with different approaches being suggested. A large part of this work is based on game theory and attempts to find equilibria in the presence of attackers and defenders of a certain skill or optimize defense and attack strategies. These approaches comprise Stackelberg Games [1,38], Zero-Determinant Theory [40],

Table 6 Comparative overview of average simulation time

Defense	Average time (s)
No defense	46
VM resetting	131
Live migration	166
IP shuffling	220
Cold migration	284
Overall	169

empirical game-theoretic analysis [36], Markov modeling [18,27,30,41,42], as well as stochastic petri nets [7,14]. However, none of them attempt to determine a given technique's real effects. Instead, intended effects are assumed and evaluation focuses on finding effective strategies to employ these techniques. While these are ultimately needed to correctly utilize respective defenses, questions on their actual effects remain unanswered. On the other hand, there is also research investigating whether a defensive technique contributes to security in the first place. Proposals range from mathematical formalization [16,17] through modeling and simulation [24] to real-world testbeds [15,37]. Yet, these either focus on only small sections of larger networks, thus potentially neglecting environmental factors, or require too much effort to consider numerous techniques and scenarios as is the case with testbeds. Given that our findings on VM migration performance contradict the results from previous research, we deliberately chose to have a closer look at evaluation approaches that also consider VM migration.

In 2016, Hong and Kim introduced their hierarchical attack representation model (HARM) [24] to address the challenges of evaluating MTD techniques at the example of VM *live migration*. This multi-layered approach is based on graphical security models such attack trees and graphs, yet overcomes their limitations with regard to incorporating adaptations that MTD ultimately requires. While it is not made clear if defenses are modeled by their intended effect or underlying mechanisms, results indicate that the scheme is capable to detect security improvement and deterioration. However, the scenario used for evaluating effectiveness consists of only three physical hosts and five VMs where migrating specific machines indeed improves security due to a sub-optimal initial state and, most importantly, knowledge about the attacker's current progress. While this can be considered a realistic setting, there are numerous equally realistic settings where the attacker's progress is not known and VM migration may cause transition to an insecure state. Additionally, it is assumed that no negative effect may result from co-location of VMs. Considering the track record of vulnerabilities in common hypervisors, and the fact that most of them default to attaching guest OSes to the same

virtual switch (e.g. Xen), this is a strong assumption. In consequence, Hong and Kim conclude that VM migration is effective to improve security. Further defense evaluation conducted using HARM [4,5] revealed that migration may increase the number of attack paths, yet not leading to any security degradation as *VM migration* would still fend these off. Instead, results indicate that the overall risk and the “return on attack” was ultimately lowered. Enoch et al. [21] present a further developed version of the HARM that incorporates temporal aspects.

The evaluation approach of Debroy et al. [19] is also concerned with VM *live migration*, yet not to assess its actual effects but to optimize its utilization in a defense strategy to fend off distributed denial of service (DDoS) attacks. The authors determine performance indicators such as cost of migration related to resource consumption and service degradation for legitimate users that ought to be minimized. At the same time migration should be timed so that DDoS attacks are prevented while choosing optimal migration locations that take resource utilization efficiency into account. Analysis to derive these timings is based on the common assumption that DDoS attacks can be modeled as a Poisson process. Should proactive migration not prevent a DDoS attack, reactive migration is suggested based on alerts from an intrusion detection system (IDS). Experiments are performed on an SDN-enabled GENI Cloud and results indicate that an optimized defense strategy can reduce the success rate of DDoS attacks by up to 40% while reducing cost as compared to periodic migration. Since the threat model is limited to DDoS attacks, downsides that may result from co-location of VMs are not accounted for. However, Debroy *et al.* consider service interruption and additional load to be impediments of VM migration and include these in their optimization problem.

An earlier approach to evaluating VM migration is presented by Wang et al. [39]. The suggested framework primarily considers cost of performing defense actions as opposed to cost of being attacked, as well as the effort required by an attacker to successfully compromise a system. The cost incurred to the defender for migrating a VM are derived from the mean downtime of services and related monetary net loss. Cost of being attacked include damages from loss of confidentiality, integrity and availability but also reputation. Together with historic information on attacks and knowledge of the mean effort required to conduct them, distribution fitting is done to enable prediction of future attacks. Based on this, the framework optimizes for reducing cost in the long run while keeping migration intervals shorter than attack intervals. Interestingly enough, the authors consider covert channel attacks that may result from co-location of VMs and use this to motivate migration in the first place. However, the fact that migration may enable

the attacker’s VM to attack even more other machines is not considered.

Other approaches that evaluate MTD-related cost in form of resource consumption and service degradation, yet not focusing on VM migration, have been presented by Connel et al. [18], Chen et al. [12], and Mendonça et al. [32]. The simulation-based approaches presented by Zhuang et al. [44–46] also consider defense-related service disruption, yet not as cost to be minimized, but as functional and security requirements that must not be violated by a defense. The latest survey from Cho et al. [13] provides a comprehensive summary on MTD techniques, the attacks they are supposed to defend against, as well as metrics and approaches to evaluation. However, none of the articles referenced therein addresses potential security degradation related to VM migration or other types of movement for that matter. Bajic and Becker recently made the case in a position paper [9] that there is a lack of attention to such negative effects of movement in MTD research that should be addressed in the future.

7 Conclusion and future work

In this work we presented an attack simulation framework that is able to quantitatively and qualitatively evaluate and compare different network defense techniques at a larger scale. Prior experiments had shown that simulation is generally capable of yielding meaningful insights. Yet, they also emphasized that creating a range of realistic benchmark networks is crucial for a fair evaluation as seemingly small changes in a network may have a considerable impact on simulation results. However, manually modeling benchmark networks is prone to bias and, above all, time-consuming. To solve this issue, the framework implements “benchmark fuzzing”, allowing to automatically generate large numbers of realistic benchmark networks from a single scenario definition. This definition might dictate the basic network structure, for example, while declaring other characteristics subject to probability. Networks generated from this definition will follow this prescribed structure, yet differ in aspects such as the number of nodes, deployed applications or any other parameter.

As a case study we created a scenario definition for a mid-sized corporate network and used it to automatically generate 200 benchmark network instances. As defenses we used two forms of VM migration (i.e., *live and cold migration*), *IP shuffling* and *VM resetting*, that have previously only been analyzed in manually tweaked networks. Two new metrics have been introduced to quantify the security implications of the defenses under test. The first metric indicates how far an attacker can infiltrate the network for a given defense by measuring gained revenue, while the sec-

ond metric exposes throttling effects on attacks. In summary, the analysis of simulation results not only confirmed previous findings indicating potential security degradation caused by VM migration, but also showed that this effect is not incidental. Furthermore, evaluation generated new insights concerning the effects of other defenses and the frequencies at which these occur. In particular, analysis of the overall 100,000 independent simulations revealed that VM migration reduced maximum attacker revenue in only 3% of the cases, while increasing it in more than a third. In contrast, *IP shuffling* and *VM resetting* never caused an increase of r_{max} but a reduction in 2.5%, respectively 4.5% of the cases. Furthermore, both types of VM migration also caused a higher average accumulated revenue, indicating the speed at which revenue was acquired. This was observed in 43% and 30.5% of the cases for *live and cold migration*, respectively. Yet, it should not go unnoticed that VM migration reduced the average accumulated revenue on other occasions. For *live migration*, this was observed in 13.5% of the cases, for *cold migration* in 37.5% of the cases. Again, *IP Shuffling* and *VM resetting* performed best, not causing any increase but only a decrease of r_{avg} in 23% and 40.5% of the cases.

The in-depth analysis of two exemplarily chosen networks provided further qualitative insights, helping to understand how figures from the quantitative analysis came into existence. This was particularly relevant to make sense of the contradictory effects observed for VM migration. Investigation revealed that VM migration poses the risk of opening up attack paths that would not have existed otherwise, resulting from changed co-location of VMs and adaptations in connectivity to accommodate VMs in their new location. However, benchmark networks exhibiting a high number of viable attack paths benefited from VM migration for the very same reasons, causing the observed positive effects by reducing the attacker's options. This emphasizes the need to extend testing across larger numbers of diverse benchmark networks which we achieved through benchmark fuzzing, allowing to detect and distinguish dominant effects and incidental corner cases. It should be noted that presented results, despite the use of fuzzing, must be considered in the context of the investigated scenario. The benchmark networks in our case study represented variations of a certain type of network, allowing for the observation of certain effects. However, crafting scenario definitions that describe inherently different networks may yield other defense effects and frequencies at which these occur.

7.1 Future work

To obtain meaningful simulation results, detailed and realistic modeling of scenarios is key. Developing these for different targets such as data centers, large enterprise networks, and IoT environments are important next steps.

Ideally, such scenarios are not only used by the researchers who developed them, but are employed by others as well, serving as common benchmarks to ultimately unify evaluation and enable the fair comparison of defense techniques across experiments conducted by different research groups. However, considering the growing number of proposed and developed defense techniques, broadening the range of modeled attacks and defenses to extend evaluation beyond the selected MTD techniques presented in this work, is equally relevant.

Besides realistic modeling of benchmark scenarios and defenses, working on realistic attacker capabilities is an important topic. In our analysis we used a greedy and powerful attacker who pursues all available attack avenues. For the considered defenses this was appropriate as attacker actions only resulted in beneficial effects from an attacker's perspective. However, to include any kind of intrusion detection systems in the analysis, we need to model intelligent attackers who try to avoid being detected. An intelligent attacker would also help to analyze defenses that are based on deception. The benchmark fuzzing approach introduced in this work to automatically generate a large number of related but different benchmark networks is already an important step towards training such an attacker. Manual benchmark modeling would be too cumbersome to generate enough training data for most artificial intelligence (AI) algorithms to be effective. Attack simulations with such intelligent attackers could not only be used to further evaluate defenses but also improve defense deployment by optimizing honeypot placement, for example. Hence, combining the attack simulation approach with an AI-enabled attacker is a promising and interesting direction for future work.

Supplementary Information The online version contains supplementary material available at <https://doi.org/10.1007/s10207-021-00552-9>.

Funding This study is funded by Rheinmetall and was conducted while the authors were with ESMT Berlin.

Availability of data and material Supplemental material is provided with this submission. It includes all boxplots as described in the experimental result section and the associated simulation data as pickle files (Online Resource 2).

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Code availability Python scripts are included to recalculate all boxplots, histograms, and tables of the simulation results, as well as the chart depicting the distribution of dominant effect combinations (Online

Resource 2). Code for scenario translation and simulation engine is not included (copyright of the code lies with our industry partner Rheinmetall). The code in the high-level language describing the case study scenario as well as the additional smaller scenario developed for performance analysis together with attacker and defender actions is included (Online Resource 1).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

A Implementation

The following sections cover implementational aspects of benchmark network generation and simulation.

A.1 Benchmark network generation

The benchmark network generator transforms high-level scenario definitions written in the modeling language into low-level representations in Prolog. It is implemented in Python 3.7 and relies on the Parsimonious library³ from Erik Rose, a parser based on parsing expression grammars (PEG). Processing of all files that are included in, or referenced by the scenario definition is done in a multi-stepped approach to resolve different statements and instructions in a convenient order. Nested calls to templates (that serve as blue prints for more complex elements) are processed recursively as these may occur at any depth of the structure described by the scenario definition and the referenced templates. This recursive processing creates an instance of the main *worker* for every element instantiation, triggered by the *new* keyword followed by the respective template name. Instances of the *worker* class process complete files in consideration of provided contextual information which may comprise parameters to initialize said element, as well as the identifier of the parent element from which instantiation of the subordinate element has been initiated. The first *worker* instance that processes the main scenario definition file is the only one that operates without any contextual information since this file represents the root of the scenario. Once a file has been fully loaded, the parser evaluates it using a grammar that specifies symbols and defines the patterns

Fig. 8 Traversal of the abstract syntax tree to generate benchmark network from scenario definition

to recognize these. Said grammar is formally described in extended Backus-Naur form (EBNF) that covers all symbols the parser might encounter. Listing 6 provides an extract of the EBNF used to process scenario definitions. There, atomic symbols such as *uppercase* and *lowercase* are defined as character sequences from A to Z and a to z, respectively. These, in turn, are used to form more complex symbols such as *type* and *identifier*, for example, that are subsequently used to compose complete expressions like the *declaration* of an element.

```

1  declaration = ws type ws
   identifier eos
2  ws = " " *
3  type = lowercase alphanum*
4  identifier = lowercase alphanum*
5  eos = ws ";" ws
6  lowercase = ~"[a-z]"
7  uppercase = ~"[A-Z]"
8  alphanum = digit / letter
9  digit = ~"[0-9]"
10 letter = uppercase / lowercase

```

Listing 6 EBNF extract for parsing element declarations

If a document has been parsed successfully, an abstract syntax tree (AST) is generated. This is a hierarchical representation of all identified symbols and how they are related to allow for subsequent processing. This tree is traversed bottom up by a node visitor, starting with the leaves. In this course, identified symbols are processed, transformed if needed, and returned to their parent node. To better illustrate this, Figure 8 depicts a simplified AST that relates to symbols introduced in the previously mentioned sample EBNF. While the single ellipses represent identified symbols, the arrows indicate the processing order during traversal of the tree. In the case of language features that serve to diversify derived benchmark networks, the processing of related symbols in the AST will randomly determine a specific value based on a set of given options and chances, and use it during the remainder of benchmark network generation. This may include utilization in if-clauses, thus potentially opening up completely new branches of the scenario definition, yielding the high degree of diversity.

³ <https://github.com/erikrose/parsimonious>.

Fig. 9 Simplified processing scheme of the simulation engine and involved components

A.2 Simulation engine

The simulation engine utilizes the generated benchmark networks to perform interaction simulation of all involved actors, apply resulting state changes accordingly, and measure progress in terms of acquired revenue. It is implemented in Python (3.7) and SWI-Prolog, one of the most actively maintained open-source Prolog implementations. To bridge these two, the PySwip library⁴ is used that allows to perform queries to Prolog databases from within Python programs. To accomplish this, the Python-based part of the simulator maintains a dedicated Prolog instance that has been loaded with the benchmark network. For the sake of simplicity, Python takes care of all organizational aspects that are related to the simulation process. This includes keeping track of rounds, durations, and probabilities, but also logging all interaction, no matter if successful or not. Furthermore, Python also handles queuing and execution of actions and only resorts to Prolog to determine if these are actually feasible by verifying fulfillment of their requirements. If, and only if, these are fulfilled at the time of attempted execution, state changes will be applied. This again is done using Prolog to ensure that subsequent reasoning about feasible actions is based on the benchmark network's actual state. How this relates to the simulation process described in Sect. 2.5 is depicted in Fig. 9, showing a simplified version of the involved steps.

To guarantee compatibility of the Python framework and the Prolog instance, benchmark networks exhibit a defined structure, allowing the python scripts to retrieve any kind

of information and execute any kind of action in a uniform way. As a result, the simulation engine is fully independent from what has been modeled so that scenario definitions may easily be extended or changed without affecting operations of the simulator. To increase performance of the reasoning process, that is highly dependent on the number of interconnected nodes and the quantity of available actions, so called *tabling* is used. When activated, this feature takes care to store the results of all performed reasoning operations as to look them up when needed again, instead of recomputing them. While such information obviously becomes invalid, once a state change occurred and must therefore be deleted upon every successfully performed action, it considerably improves performance of finding all of an actor's options in any given round.

B Additional data on coverage

Section 4.3 presented the most dominant observable effects and their frequency of occurrence among the 200 benchmark networks. While these already made for a large proportion of investigated networks, a complete list of all observed effects is presented in Table 7.

Table 7 Overview of observed defense performance for *cold migration* (C), *live migration* (L), *VM resetting* (R), and *IP shuffling* (I) with '+' indicating a larger, '-' a smaller attacker revenue

rmax CLRI	ravg CLRI	count	rmax CLRI	ravg CLRI	count
====	====	49	====	++-	1
++==	++==	37	====	--	1
====	--	17	-+--	--=-	1
====	----	13	----	--	1
====	---=	9	++==	----	1
====	--	8	----	++	1
====	-+=	8	++=-	++==	1
++==	++=-	8	++==	==	1
++==	+=	6	----	--	1
++==	++	6	++==	----	1
====	+=-	4	++==	---+	1
====	--	3	==	+=	1
====	+=	2	-+=	--=-	1
++==	---=	2	++==	-+=	1
====	++==	2	====	====	1
++==	-+=	2	-+=	----	1
====	++=-	2	++==	++=-	1
++-	++	2	++==	+=-	1
---=	--	2			

⁴ <https://github.com/yuce/pyswip>.

In this table, every observed effect combination is characterized by eight measures representing the impact that each of the four defenses had on both $rmax$ and $ravg^m$. This impact can either be an increase represented through a plus sign (+) or a decrease represented through a minus sign (−). Cases where no significant impact was measured are represented through an equal sign (=). Note that impact is always measured against $rmax$ and $ravg^m$ in the presence of *no defense* for a given benchmark network. Different networks exhibiting the same combination of defense effects are put into the same bin.

As can be seen in Table 7, this “encoding” of effects allows for a relatively compact summary of all observed combinations. In this representation, the most frequently made observation that none of the defenses has any measurable impact, neither on $rmax$ nor on $ravg^m$, is characterized by equal signs for all eight measures with a count of 49. Similarly, the second most dominant effect of both *cold and live migration* increasing $rmax$ and $ravg^m$, while *VM resetting* and *IP shuffling* have no impact, is represented as a combination of plus and equal signs with a count of 37. The remaining effect combinations are presented using the same scheme, appearing in descending order.

References

1. Abdel Wahab, O., Bentahar, J., Otok, H., Mourad, A.: Resource-aware detection and defense system against multi-type attacks in the cloud: repeated Bayesian stackelberg game. *IEEE Trans. Depend. Secure Comput.* **1** (2019)
2. Ahmed, N.O., Bhargava, B.: Bio-inspired formal model for space-time virtual machine randomization and diversification. *IEEE Trans. Cloud Comput.* **1** (2020)
3. Ahmed, N.O., Bhargava, B.: Mayflies: a moving target defense framework for distributed systems. In: *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, pp. 59–64. MTD '16, ACM (2016)
4. Alavizadeh, H., Jang-Jaccard, J., Kim, D.S.: Evaluation for combination of shuffle and diversity on moving target defense strategy for cloud computing. In: *2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 573–578 (2018)
5. Alavizadeh, H., Kim, D.S., Jang-Jaccard, J.: Model-based evaluation of combinations of shuffle and diversity MTD techniques on the cloud. *Future Gener. Comput. Syst.* **111**, 507–522 (2019)
6. Almohri, H.M.J., Watson, L.T., Evans, D.: Misery digraphs: delaying intrusion attacks in obscure clouds. *IEEE Trans. Inf. Forensics Secur.* **13**(6), 1361–1375 (2018)
7. Anderson, N., Mitchell, R., Chen, I.R.: Parameterizing moving target defenses. In: *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–6 (2016)
8. Bajic, A., Becker, G.T.: Attack simulation for a realistic evaluation and comparison of network security techniques. In: *Nordic Conference on Secure IT Systems*, pp. 236–254. Springer (2018)
9. Bajic, A., Becker, G.T.: A critical view on moving target defense and its analogies. In: *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pp. 277–283 (2020)
10. Bangalore, A.K., Sood, A.K.: Securing web servers using self cleansing intrusion tolerance (SCIT). In: *2009 Second International Conference on Dependability*, pp. 60–65 (2009)
11. Chang, S., Park, Y., Ashok Babu, B.B.: Fast IP hopping randomization to secure hop-by-hop access in SDN. *IEEE Trans. Netw. Serv. Manag.* **16**(1), 308–320 (2019)
12. Chen, Z., Chang, X., Han, Z., Yang, Y.: Numerical evaluation of job finish time under MTD environment. *IEEE Access* **8**, 11437–11446 (2020)
13. Cho, J., Sharma, D.P., Alavizadeh, H., Yoon, S., Ben-Asher, N., Moore, T.J., Kim, D.S., Lim, H., Nelson, F.F.: Toward proactive, adaptive defense: a survey on moving target defense. *IEEE Commun. Surv. Tutor.* **22**(1), 709–745 (2020)
14. Cho, J.H., Zhu, M., Singh, M.: Modeling and analysis of deception games based on hypergame theory. In: *Autonomous Cyber Deception*, pp. 49–74. Springer (2019)
15. Chowdhary, A., Alshamrani, A., Huang, D., Liang, H.: MTD analysis and evaluation framework in software defined network (MASON). In: *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization. SDN-NFV Sec '18*, pp. 43–48. ACM (2018)
16. Connell, W., Menasce, D.A., Albanese, M.: Performance modeling of moving target defenses with reconfiguration limits. *IEEE Trans. Depend. Secure Comput.* **1** (2018)
17. Connell, W., Albanese, M., Venkatesan, S.: A framework for moving target defense quantification. In: *IFIP International Conference on ICT Systems Security and Privacy Protection*, pp. 124–138. Springer (2017)
18. Connell, W., Menascé, D.A., Albanese, M.: Performance modeling of moving target defenses. In: *Proceedings of the 2017 Workshop on Moving Target Defense. MTD '17*, pp. 53–63. ACM (2017)
19. Debroy, S., Calyam, P., Nguyen, M., Neupane, R.L., Mukherjee, B., Eeralla, A.K., Salah, K.: Frequency-minimal utility-maximal moving target defense against DDoS in SDN-based systems. *IEEE Trans. Netw. Serv. Manag.* **17**(2), 890–903 (2020)
20. Dunlop, M., Groat, S., Urbanski, W., Marchany, R., Tront, J.: MT6D: a moving target IPv6 defense. In: *Military Communications Conference—MILCOM 2011*, pp. 1321–1326 (2011)
21. Enoch, S.Y., Hong, J.B., Ge, M., Alzaid, H., Kim, D.S.: Automated security investment analysis of dynamic networks. In: *Proceedings of the Australasian Computer Science Week Multiconference. ACSW '18*. ACM (2018)
22. Han, Y., Chan, J., Alpcan, T., Leckie, C.: Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Trans. Depend. Secure Comput.* **14**(1), 95–108 (2015)
23. Holm, H., Shahzad, K., Buschle, M., Ekstedt, M.: P² CySeMoL: predictive, probabilistic cyber security modeling language. *IEEE Trans. Depend. Secure Comput.* **12**(6), 626–639 (2015)
24. Hong, J.B., Kim, D.S.: Assessing the effectiveness of moving target defenses using security models. *IEEE Trans. Depend. Secure Comput.* **13**(2), 163–177 (2016)
25. Johnson, P., Vernotte, A., Ekstedt, M., Lagerström, R.: pwnpr3d: an attack-graph-driven probabilistic threat-modeling approach. In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pp. 278–283. IEEE (2016)
26. Kewley, D., Fink, R., Lowry, J., Dean, M.: Dynamic approaches to thwart adversary intelligence gathering. In: *DARPA Information Survivability Conference and Exposition II, 2001. DISCEX '01. Proceedings*, vol. 1, pp. 176–185 (2001)
27. Lei, C., Zhang, H.Q., Wan, L.M., Liu, L., He Ma, D.: Incomplete information Markov game theoretic approach to strategy generation for moving target defense. *Comput. Commun.* **116**, 184–199 (2018)

28. Li, J., Yackoski, J., Evancich, N.: Moving target defense: a journey from idea to product. In: *Proceedings of the 2016 ACM Workshop on Moving Target Defense. MTD '16*, pp. 69–79. ACM (2016)
29. MacFarland, D.C., Shue, C.A.: The SDN shuffle: Creating a moving-target defense using host-based software-defined networking. In: *Proceedings of the Second ACM Workshop on Moving Target Defense, MTD '15*, pp. 37–41. ACM (2015)
30. Maleki, H., Valizadeh, S., Koch, W., Bestavros, A., van Dijk, M.: Markov modeling of moving target defense games. In: *Proceedings of the 2016 ACM Workshop on Moving Target Defense. MTD '16*, pp. 81–92. ACM (2016)
31. McGill, R., Tukey, J.W., Larsen, W.A.: Variations of box plots. *Am. Stat.* **32**(1), 12–16 (1978)
32. Mendonça, J., Cho, J.H., Moore, T.J., Nelson, F.F., Lim, H., Zimmermann, A., Kim, D.S.: Performability analysis of services in a software-defined networking adopting time-based moving target defense mechanisms. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing. SAC '20*, p. 1180–1189. ACM (2020)
33. Narantuya, J., Yoon, S., Lim, H., Cho, J., Kim, D.S., Moore, T., Nelson, F.: SDN-based IP shuffling moving target defense with multiple SDN controllers. In: *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks—Supplemental Volume (DSN-S)*, pp. 15–16 (2019)
34. Neupane, R.L., Neely, T., Chettri, N., Vassell, M., Zhang, Y., Calyam, P., Durairajan, R.: Dolus: Cyber defense using pretense against DDoS attacks in cloud platforms. In: *Proceedings of the 19th International Conference on Distributed Computing and Networking. ICDCN '18*, pp. 30:1–30:10. ACM (2018)
35. Ou, X., Govindavajhala, S., Appel, A.W.: Mulval: a logic-based network security analyzer. In: *USENIX Security Symposium*, p. 8. Baltimore, MD (2005)
36. Prakash, A., Wellman, M.P.: Empirical game-theoretic analysis for moving target defense. In: *Proceedings of the Second ACM Workshop on Moving Target Defense. MTD '15*, pp. 57–65. ACM (2015)
37. Taylor, J., Zaffarano, K., Koller, B., Bancroft, C., Syversen, J.: Automated effectiveness evaluation of moving target defenses: Metrics for missions and attacks. In: *Proceedings of the 2016 ACM Workshop on Moving Target Defense. MTD '16*, pp. 129–134. ACM (2016)
38. Vadlamudi, S.G., Sengupta, S., Taguinod, M., Zhao, Z., Doupe, A., Ahn, G.J., Kambhampati, S.: Moving target defense for web applications using Bayesian stackelberg games: (extended abstract). In: *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems. AAMAS '16*, pp. 1377–1378 (2016)
39. Wang, H., Li, F., Chen, S.: Towards cost-effective moving target defense against DDoS and covert channel attacks. In: *Proceedings of the 2016 ACM Workshop on Moving Target Defense. MTD '16*, pp. 15–25. ACM (2016)
40. Wang, S., Shi, H., Hu, Q., Lin, B., Cheng, X.: Moving target defense for internet of things based on the zero-determinant theory. *IEEE Internet Things J.* **7**(1), 661–668 (2020)
41. Xiong, X., Yang, L., Zhao, G.: Effectiveness evaluation model of moving target defense based on system attack surface. *IEEE Access* **7**, 9998–10014 (2019)
42. Zheng, J., Siarni Namin, A.: Enforcing optimal moving target defense policies. In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 753–759 (2019)
43. Zheng, J., Namin, A.S.: A survey on the moving target defense strategies: an architectural perspective. *J. Comput. Sci. Technol.* **34**(1), 207–233 (2019)
44. Zhuang, R., Bardas, A.G., DeLoach, S.A., Ou, X.: A theory of cyber attacks: a step towards analyzing MTD systems. In: *Proceedings of the Second ACM Workshop on Moving Target Defense. MTD '15*, pp. 11–20. ACM (2015)
45. Zhuang, R., DeLoach, S.A., Ou, X.: A model for analyzing the effect of moving target defenses on enterprise networks. In: *Proceedings of the 9th Annual Cyber and Information Security Research Conference. CISR '14*, pp. 73–76. ACM (2014)
46. Zhuang, R., Zhang, S., DeLoach, S.A., Ou, X., Singhal, A.: Simulation-based approaches to studying effectiveness of moving-target network defense. In: *National Symposium on Moving Target Research. NIST* (2012)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.