Specification-driven Moving Target Defense Synthesis

Md Mazharul Islam, Qi Duan and Ehab Al-Shaer
University of North Carolina Charlotte
Charlotte, North Carolina, USA
{mislam7,qduan,ealshaer}@uncc.edu

ABSTRACT

Cyber agility enables cyber systems to defend proactively against sophisticated attacks by dynamically changing the system configuration parameters (called mutable parameters) in order to deceive adversaries from reaching their goals, disrupt the attack plans by forcing them to change their adversarial behaviors, and/or deterring them through prohibitively increasing the cost for attacks. However, developing cyber agility such as moving target defense techniques that are provable safe is a highly complex task that requires significant time and expertise. Our goal is to address this challenge by providing a framework for automating the creation of configuration-based moving target techniques rapidly and safely.

In this paper, we present a cyber agility synthesis framework, called MTDSynth, that contains a formal ontology, MTD policy language, and MTD controller synthesis engine for implementing configuration-based moving target defense techniques. The policy language contains the agility specifications required to model the MTD technique, such as sensors, mutation trigger, mutation parameters, mutation actions, and mutation constraints. Based on the mutation constraints, the MTD controller synthesis engine provides an MTD policy refinement implementation for SDN configuration with provable properties using constraint satisfaction solvers. We show several examples of MTD controller synthesis, including temporal and spatial IP mutation, path mutation, detector mutation.

We developed our ActivSDN over OpenDaylight SDN controller as an open programming environment to enable rapid and safe development of MTD sense-making and decision-making actions. Our implementation and evaluation experiments show not only the feasibility of MTD policy refinement but also the insignificant computational overhead of this refinement process.

CCS CONCEPTS

• **Software and its engineering** → *Syntax*; *Semantics*; • **Networks** → *Network manageability*; *Network privacy and anonymity*; *Network management*; *Formal specifications.*

ACM Reference Format:
Md Mazharul Islam, Qi Duan and Ehab Al-Shaer. 2019. Specification-driven Moving Target Defense Synthesis. In *6th ACM Workshop on Moving Target Defense (MTD'19), November 11, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338468.3356830

MTD'19, November 11, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6828-5/19/11…$15.00
https://doi.org/10.1145/3338468.3356830

1 INTRODUCTION

Cyber defense techniques are mostly non-adaptive and take a long time to detect and respond (hours, days even months). Moreover, the defense techniques are rigid and do not provide agility capability to mitigate threat proactively. In addition, the static and predictable behavior of cyber systems from the attackers' view creates a fundamental design vulnerability.

Cyber agility allows the cyber system to defend proactively against a wide-scale vector of sophisticated attacks by dynamically changing the system parameters and defense strategies in a timely and economical fashion. It can provide robust defense by deceiving attackers from reaching their goals, disrupting their plans via changing adversarial behaviors, and deterring them through prohibitively increasing the cost for attacks. However, developing agility on cyber defense is a highly complex task, and it requires significant effort in implementation and management in order to obtain a safe and well-orchestrated MTD operations. As a result, few MTD techniques are developed and validated on the real-life operational environment. To tackle this problem, we developed a cyber agility framework, called MTDSynth, that allows MTD developers for creating MTD control programs using a high-level cyber agility policy language (HAPL) that provides the required constructs for configuration-based MTD techniques including the following: (1) *mutation triggers* which can be time-based or event-based using user-defined network *sensors*, (2) MTD *mutable system parameters* that will be dynamically changed based on the trigger, (3) *configuration parameters* that dependent on the mutable parameters, (4) *mutation functions* and *mutation constraints* that dictates the methodology to compute and optimize the selection of new mutation value, (5) *mutation attributes* that can be used to define the mutation scope or domain. MTDSynth also provides a policy refinement engine to synthesize the control program using Software-defined networking (SDN)[2] controller and OpenFlow[20] configurations. As a result, an MTD policy that is instrumented by user-defined function can be automatically translated to a verified MTD control programs that satisfy the constraints and properties defined in the agility policy specification.

We used SDN as a platform for our MTD synthesis because it provides a robust mechanism for dynamic and disruptive network management. SDN provides programmatic ability into network configurations while monitoring the whole network. The fundamental facility achieved from SDN is to manage the network configuration dynamically from a central controller for quick response and diagnosis. We developed an extensible API interface, called ActiveSDN, over OpenDaylight SDN controller[21] to facilitate the synthesis process. ActiveSDN supports the implementation of the sensors and MTD actions defined in the HAPL at a high level without the need to focus on any low-level OpenFlow configuration. In addition, ActiveSDN incorporates the Satisfiability Modulo Theory

(SMT) constraints satisfiability solver [11], to optimize the MTD and agility actions (selection of parameters and configurations values) at real-time. We also show several examples of defining various configuration-based MTD techniques using different parameters such as temporal and spatial RHM[7], RRM [13], detector mutation [27].

Thus, MTDSynth provides an open programming environment to develop rapidly and safely sense-making and decision making MTD actions to enable cyber agility for dynamic cyber defense capabilities on ActiveSDN using OpenDaylight controller. We evaluated MTDSynth performance to assess the inherent computational overhead due to the multi-layer synthesis process. We compared the implementation of MTDSynt for the temporal RHM, spatial RHM, and RRM with the native SMT implementation of these MTD techniques. We found that for temporal RHM the average configuration delay is very small (0.027s) and is unaffected by the network size, and for spatial RHM and RRM the average configuration delay is in the order of tens of milliseconds to hundreds of milliseconds, which validates the feasibility and scalability of our framework.

The paper is organized as follows: Section 2 describe the background and related works about common MTD techniques; Section 3 presents the MTD policy language specification with examples; Section 4 present the architecture of the MTDSynth framework; Section 5 presents case studies about different types of MTD techniques created by MTDSynth and their deployment into the network; Section 6 describes the implementation of MTDSynth and evaluate the performance of the framework against various MTD techniques; Section 7 concludes our work with future aspects.

2 BACKGROUNDS AND RELATED WORKS

In this paper we develop a general framework to instantiate any MTD technique from the user defined specifications. To illustrate our framework, we will use the following existing MTD techniques as examples: (1) Temporal Random Host IP Mutation (Temporal RHM) [7], (2) Spatial Random Host IP Mutation (Spatial RHM) [16], (3) Random Route Mutation (RRM) [13], and (4) MTD to disrupting stealthy bots [27].

Temporal RHM. Temporal RHM can turn end-hosts into untraceable moving targets by mutating their IP addresses in an intelligent and unpredictable fashion without sacrificing network integrity, manageability or performance. In RHM, moving target hosts are assigned virtual IP addresses that change randomly and synchronously in a distributed fashion over time without disrupting active connections.

Spatial RHM. In spatial RHM, to reach each destination host h_j, each source host h_i is associated with an ephemeral IP (eIP), such that this eIP could be only used by h_i to reach h_j. The distribution based on which these new mappings are determined can be either *uniform* or *deceptive (adversary-adaptive)*. The mutation uses a strategy selection algorithm to determine the appropriate way at any given time by analyzing the behavior of potential adversaries in the network.

Random Route Mutation (RRM). RRM allows for switch routes in the network periodically or based on feedback from network monitors. The main goal of RRM is to change the route between a given source and destination address randomly to disable the attack capabilities to launch an effective eavesdropping or DoS attacks on the specific node or link in the route.

Disrupting Stealthy Bots. Disrupting Stealthy Bots is a MTD approach for placing detectors across the network in a resource constrained environment and dynamically and continuously changing the placement of detectors over time to defend against stealthy bots.

The notion of mutable networks as a frequently randomized changing of network addresses and responses was initially proposed in [4]. The idea was later extended as part of the MUTE network which implemented the moving target through random address hopping and random fingerprinting [5].

Existing IP mutation techniques include Dynamic Network Address Translation (DYNAT) [18, 22, 23], Applications that Participate in their Own Defense (APOD) [8], Address Routing Gateway (ARG) [25], Network Address Hopping (NAH) [26], Random Host IP Mutation (RHM) [7], OpenFlow Random Host IP Mutation (OF-RHM) [15], etc.

DYNAT is a technique developed to dynamically reassign IP addresses to confuse any would-be adversaries sniffing the network. They obfuscate the host identity information (IP and Port) in TCP/IP packet headers by translating the identity information with preestablished keys. BBN ran series of red-team tests to test the effectiveness of DYNAT, while Sandia's DYNAT report [22, 23] examines many of the practical issues for DYNAT deployment.

Spatio-temporal Address Mutation (STORM) [16] can defend against collaborative scanning worm and APT attacks. It can distort attackers' view of the network by causing the collected reconnaissance information to expire as adversaries transition from one host to another or if they stay long enough in one location.

The work in [13] provided a general formalization for RRM with various operational and QoS constraints. The route selection is random and the new constraints can be added conveniently. In practical networks, the number of disjoint paths is usually very small [28], so the work in [13] analysed the MPE with non-disjoint paths for RRM. The work in [12] presented a cyber deception framework, called CONCEAL, as a composition of mutation, anonymity, and diversity to maximize key deception objectives.

Developing cyber agility framework is a complex task because of the automatic yet fast orchestration and management of network configuration without breaking the mission integrity [9, 19]. Agility framework requires comprehensive metrics for the safe deployment of mitigation techniques [24].

3 AGILITY POLICY LANGUAGE SPECIFICATION

The goal of MTDSynth is twofold: 1) an MTD language specification and 2) an Engine that provides a programmable environment for MTD policy creation leveraging the language. The language specification will allow people to program MTD techniques in high-level and the Engine will deploy that technique into the network *safely* by orchestrating low-level network configurations (e.g., DNS

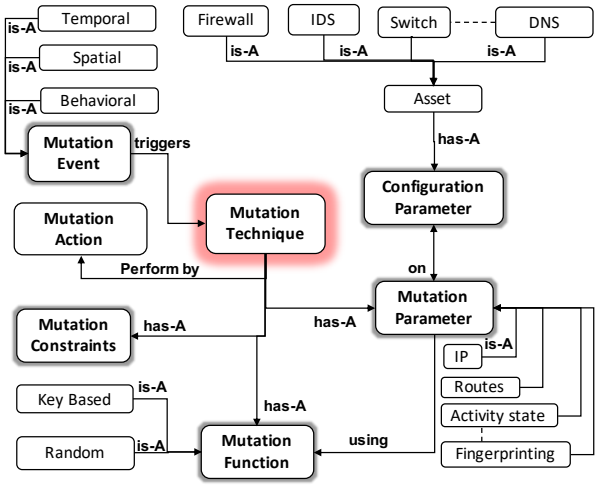


Figure 1: Cyber Agility Policy Ontology for MTDSynth.

record, flow rules in switch tables) automatically. By *safely*, it means that the deployment of MTD techniques doesn't violate the mission integrity as the automatic configuration of such sophisticated techniques can jeopardize the reachability, liveness, and fairness of the network communication. Therefore, MTDSynth provides minimum effort in the development of MTD techniques and puts safeguard onto it by considering the constraints. To define a fine grain MTD specification, we need an ontology that can best describe the formal syntax of MTD techniques creation. Therefore, we developed an MTD ontology and an MTD language syntax around it.

3.1 MTD ontology

Figure 1 shows the ontology for the MTD techniques. The mutation technique will be triggered by an event which can be temporal, special, or behavioral event. The temporal or spatial events are time oriented, meaning after a certain period, the mutation techniques will be triggered or continue triggering. Behavioral events depend on network behavior such as packet dropping, network scanning, link flooding, etc. Based on the event, an agility specification will be triggered. The agility specification of mutation depends on several factors like mutation parameters such as the IP address of specific hosts, configuration parameters like DNS-entry or switch flow tables, mutation functions like key-based or random distribution etc. The mutation technique get performed by the mutation actions. The event trigger time interval can initiate a periodic triggering of the mutation techniques. The ontology enforces the safety of the MTD techniques by considering the proper declaration of individual constraints that requires for each mutation strategy.

3.2 MTD Language Specification

The specifications of MTD contain the following aspects as shown in Figure 2:

MTD name N, is a *string* that can be used to define an MTD rule. Using *N*, the same rule can be used later on. Besides, the naming will help the user to keep track while creating multiple rules.

$$
\begin{aligned}
\textit{Agility Rule } \Pi &::= N:\ E \rightarrow \Lambda \\
\textit{MTD Name } N &::= \textbf{\textit{STRING}} \\
\textit{Mutation Event } E &::= \Delta \mid \alpha \\
\textit{Time Interval } \Delta &::= \textbf{\textit{NUMBER}} \\
\textit{Sensor Alert } \alpha &::= \textit{isHostScanning()} \mid \textit{isLinkFlooding()} \mid \textit{isBotDetected()} \mid \\
&\quad \textit{checkUDPICMPPRate()} \mid \textit{getAvailableBandwidth()} \mid \\
&\quad \textit{checkNewComers()} \mid \textit{checkElephantTCP()} \mid \\
&\quad \textit{getRouteLength()} \mid \textit{getCriticalLink()} \mid \textit{getRouteRisk()} \mid \\
&\quad \textit{getFlowStatistics()} \mid \textit{getAllFlowRules()} \mid \textit{getFlowRate()} \\
\textit{Agility Spec. } \Lambda &::= \textbf{\textit{MUTATE}}\ p\ id\ \textbf{\textit{OF}}\ \{attr\}\ \textbf{\textit{USING}}\ f\ \textbf{\textit{ON}}\ g\ \textbf{\textit{BY}}\ m \\
&\quad \textbf{\textit{WHILE}}\ c \\
\textit{Mutation Param. } p &::= \textbf{\textit{ROUTE}} \mid \textbf{\textit{IP}} \mid \textbf{\textit{STATE}} \\
\textit{Identifier } id &::= [A-Z] \\
\textit{Attribute } attr &::= rattr \mid ipattr \mid sattr \\
\textit{Route Attr. } rattr &::= id.src \rightarrow \textit{IPAddress}, id.dst \rightarrow \textit{IPAddress} \\
\textit{IP Attr. } ipattr &::= id.IP \rightarrow \langle list\ of\ \textit{IPAddress} \rangle \mid h \\
\textit{Host Name } h &::= \textbf{\textit{STRING}} \\
\textit{State Attr. } sattr &::= id.IP \rightarrow \langle list\ of\ \textit{IPAddress} \rangle \mid s \\
\textit{Activity Status } s &::= \textbf{\textit{UP}} \mid \textbf{\textit{DOWN}} \\
\textit{Mutation Func. } f &::= random(\eta, \eta) \mid key\text{-}based \\
\textit{Numeric } \eta &::= \textbf{\textit{NUMBER}} \\
\textit{Configuration } g &::= r \mid r, g \\
\textit{Resource } r &::= \textbf{\textit{DNS-entry}} \mid \textbf{\textit{switch-table}} \mid s \\
\textit{Mutation Action } m &::= ipMutate \mid pathMutate \mid spatalMutation \mid \\
&\quad createShadow \mid reDirect \mid migrateService \\
\textit{Constraint Spec } c &::= \beta \mid \beta\ ;\ c \\
\textit{Agility Const. } \beta &::= \alpha \mid \gamma \mid \delta \mid \delta\ op\ v \\
\textit{Mutation Const. } \gamma &::= id_{t+1}\ op\ \eta \mid id_{t+1}\ op\ id_t \mid attr_{t+1}\ op\ \eta \\
&\quad attr_{t+1}\ op\ attr_t \\
\textit{Network Func. } \delta &::= includeSwitch() \mid excludeSwitch() \mid overlap() \mid \\
&\quad canReach() \mid getAllPaths() \mid getShortestPath() \mid \\
&\quad getMinDetectionProb() \mid getAttackUncertainity() \\
\textit{Value } v &::= \eta \mid \textbf{\textit{TRUE}} \mid \textbf{\textit{FALSE}} \\
\textit{Operator } op &::= >\ \mid\ <\ \mid\ \leq\ \mid\ \geq\ \mid\ =\ \mid\ \neq\ \mid\ \cap\ \mid\ \cup\ \mid \forall \mid \exists \mid \\
&\quad +\ \mid\ -\ \mid\ \times\ \mid\ / \\
\textit{IPAddress} &::= (([0-9]|[1-9][0-9]|1[0-9]\{2\}|2[0-4][0-9]| \\
&\quad 25[0-5])\backslash.)\{3\}([0-9]|[1-9][0-9]|1[0-9]\{2\} \\
&\quad |2[0-4][0-9]|25[0-5])
\end{aligned}
$$

Figure 2: HAPL Syntax for MTDSynth.

Mutation Event E, is the trigger that initiates the mutation. The triggering event can be *time* based or network system *behavior* oriented. For example, the system admin can start IP mutation to protect critical resources in a timely fashion. Therefore after a certain period of time, the mutation technique will start and/or repeat. Here the triggering event is *time interval*, Δ. However, the admin may want to mutate a route if any of the links in the route

get flooded. In that case, the mutation triggering event is network behavior oriented, which is if a link gets flooded.

Sensor Alert α, are the sensors in ActivesSDN that can be deployed into the network to collect and measure any behavioral facts, like host scanning, link flooding, bot detecting, critical links finding, etc. The sensor primitives are implemented as functions so that the user can directly use them to trigger MTD events or create constraints for the safe deployment of MTD techniques.

Agility Specification Λ, defines the actions that are taken for the agility technique. For example, the actions of IP mutation is to mutate the IP address of the hosts in the valid address space, and the actions of route mutation are to mutate the route of the specific network flows. Therefore, the agility action specification defines to start an MTD *action* (ipMutate, pathMutate etc.) on a *parameter* (e.g., IP, route) using an mutation *function* which requires automated orchestration of network *configuration* fulfilling specific *constraints* to maintain mission integrity.

Mutation Parameters p, the mutable parameter can be IP address, route or state of an service that will be mutated over time. The mutation parameter has *attributes* that provides granularity in defining the exact mutation elements. For example, the IP parameter has attributes like host IP address and host name, the route parameter has source and destination host, the state parameter has status such as service is up or down, etc. So defining a parameter also requires proper setting of its attributes.

Mutation Function f, bounds the mutation space and defines how the mutation will happen. Providing a range in f will limit the mutation space obtained from the solution of constraints in that range. Besides, it will also used to choose the next mutation parameter from that space. MTDSynth use two mutation function from ActiveSDN, random and key-based. In random function, the mutable parameter (e.g. IP address) will be chosen in a random fashion. For key-based function, a proper hash key needs to be provided for selecting the mutation parameter.

Configuration Parameters g, include the system parameters that should be correctly configured for the mutation. Like the DNS record entry, flow rules in the switch table, etc. Note that, MTDSynth orchestrates all these low-level configurations automatically.

Mutation Action m, are the primary functionality MTDSynth uses for MTD that ActiveSDN provides. Based on an action, that particular class of MTD will be executed, for example, *ipMutate* will mutate the IP addresses, *pathMutate* will mutate the active route of a given flow, *migrateService* will dynamically change the specified services over time, etc. The user does not need to configure these actions, rather just mention the name in the policy. MTDSynth will handle the corresponding configuration of such actions. These actions are complicated to configure by hand, therefore MTDSynth will configure such actions atometically with safety.

Constraints Specification c ensures the safe deployment of the agility rule so that the mission goal remains uninterrupted. For example, while doing mutation (IP, route, or any other parameter), the reachability of the network components must not be interpreted. ActiveSDN provides a comprehensive constraint specification that can be used to define fully qualified constraints to prohibit any conflict or misconfiguration which may occur while deploying MTD rules into the network. This helps the user not to jeopardize the mission integrity while providing maximum security. To define a complete constraints specification, a series of constraints may need to be executed sequentially. This includes sensors, mutation parameter, the attributes of the parameter along with the primitives ActiveSDN API provides. To define *mutation constraints*, the current (t) attribute values will be used to determine the next ($t+1$) mutation parameter attribute values in the constraints. Therefore, a constraint specification is a combination of multiple network constraints primitives, mutation parameters, mutation parameters attributes, numerical or boolean values, that are combined with arithmetic, relational, or logical operator. MTDSynth provides a vast number of network constraints primitives from ActiveSDN API; however, user can program any constraint they want with the specification to fulfill the safety while deploying any MTD techniques.

Network Function δ are the primitives ActiveSDN provides to generate a complete constraint specification for different MTD actions. IN appendix B figure 13, we provide a detailed description of all constraints primitive. Note that, different primitives return different values, for example, *Boolean*, numeric, or list of objects. While comparing the output of such primitives, the type must be matched, otherwise, semantic error may generate in MTD policy parser module.

The primitives in MTDSynth such as *sensors*, MTD *actions* and *constraints* implemented in ActiveSDN provides the user an MTD programmable environment. Appendix B figures 12, 13, and 14 have a comprehensive list of all these primitives.

3.3 MTD Policy Examples

In this section, we will describe how MTD policy can be created using the HAPL Syntax from figure 2.

3.3.1 Route Mutation. Lets assume, user wants to mutate the route between two hosts h_1 and h_2 if any of the links in the current route of these hosts get flooded. The user has following constants: the IP address for source host h_1 is $IP_1 = 10.0.0.1$, the IP address for destination host h_2 is $IP_2 = 10.0.0.2$; deep packet inspection node = s_2. Assume there's a critical link $l(s_6, s_7)$ that is in the current route h_1 and h_2 using where l can be a target that the user wants to avoid. The agility policy for route mutation will be:

```
Path Mutation:
isLinkFlooding(l, 0.2) →
    MUTATE route R of {R.src → IP1, R.dst →IP2}
    USING random(1..N) ON switch-table BY
        pathMutate
    WHILE
        (Rt ∩ Rt+1)/Rt ≥ 0.7;
        includeSwitch(Rt+1, [s2]) = TRUE;
        excludeSwitch(Rt+1, [s6]) = TRUE;
        getRouteLength(Rt+1) ≤ 5;
        getAvailableBandwidth(Rt+1) > getFlowRate(
            IP1, IP2)×1.2;
        getRouteRisk(Rt+1) ≤ 0.25
```

R_t is the current route and R_{t+1} is the next mutated route chosen by MTDSynth following the *Agility Action*. The *isLinkFlooding(l, th)* method will check the packet drop rate in link l with the threshold th and if the rate is greater than 20%, the mutation will be triggered.

ActiveSDN has a agility primitive called *pathMutate* for the route mutation. The *pathMutate* will mutate the *data path* of a specified flow between source h_1 and destination h_2. The next mutated route R_{t+1} will be selected *randomly* from the *available route space* between h_1 and h_2. The *available route space* will be selected by solving the constrains mentioned in the *WHILE* loop. The only *configuration* changes will be done in *switch* flow rule tables. The framework will automatically orchestrate this flow rule updates in corresponding switches. To choose the next mutated route R_{t+1}, the following constraints will be executed sequentially. The constraint overlap will find the common link ratio between the current route R_t and the next chosen route R_{t+1} by intersection, *isIncludeSwitch* check whether the given route R_t contains the given switches (s_2), *excludeSwitch* ensures the given route must not contain the given switches (s_6), *getRouteLength* returns the links count in the next chosen route which helps to define a maximum hop count in a chosen route, *getAvailableBandWidth* measures the given link bandwidth and the *getFlowrate* checks the number of packets any flow (h_1, h_2) sends per second. The primitive *getRouteRisk* measures the risk of a given route that may get attacked in a probabilistic way. For example, assume that a route having n number of links, where the probability of each link get attacked is p_i where $i \in n$. Then the risk that a route will get attacked is:

$$1 - \prod_{i=1}^{n}(1 - p_i)$$

3.3.2 Spatial Mutation. Lets assume the user want to mutate the communication IP between hosts $h_1, h_2, \ldots h_m$, then the MTD policy will be:

```
Spatial IP Mutation:
isHostScanning(100, 5) →
    MUTATE IP P of {P.IP → [h1, h2, ..., hm]}
    USING random(1..N) ON DNS-entry, switch-table
        BY spatialMutation
    WHILE
        m×(m−1)/N ≤ 0.1;
        ∀i,j∈N Pt.hi ≠ Pt+1.hi
```

where N is the size of available address space, m is the number of mutating hosts. $\frac{m\times(m-1)}{N}$ is the probability of two distinct muting hosts will be assigned the same IP to reach the same destination (collision probability). The equation $\forall_{i,j\in N} P_t.h_i \neq P_{t+1}.h_i$ means that the address assigned to a communication pair will be different in two consecutive intervals.

3.3.3 Temporal IP Mutation. Lets assume, user wants to mutate IP for hosts $h_1, h_2, \ldots h_n$, where h_1 can be a web server running as *www.xyz.com*, h_2 can be a FTP server and so on. The MTD policy will be:

```
Temporal IP Mutation:
timeInterval = 5s →
    MUTATE IP P of {P.IP → [h1, h2, ..., hn]}
    USING random(1..N) ON DNS-entry, switch-table
        BY ipMutate
    WHILE
        ∀i,j∈(1,N) Pt+1.hi ≠ Pt+1.hj;
        ∀i,j∈N Pt.hi ≠ Pt+1.hi
```

Here the first equation (after the *WHILE* statement) means that at any fixed time the IP of any host in the set $\{h_1, h_2, \ldots h_n\}$ will be distinct.

The second equation means that for any host in $\{h_1, h_2, \ldots h_n\}$, the IP assigned to a host in an interval will be different from the next interval.

3.3.4 MTD against Stealthy Bots. Lets assume the user want to mutate the locations of the detecting service S, then the MTD policy will be:

```
Bot pattern detection:
isBotdetected(sig) →
    MUTATE state S of {S.location → [IP1, IP2, ...,
        IPn]}
    USING random(1..N) ON status
    WHILE
        for all t, Σ_{i=1}^{N} St.IPi < TB;
        minDetectionProb(S.location) > 0.9;
        attackUncertainty(S.location) > 0.8
```

Here service S has a set of detectors, and every detector is located in a specific host. If the status of the detector is UP then it is used for traffic sensing, and is DOWN when it is not used. Mutate the location of detectors is equivalent to change the status of the detectors. T_B is the upper limit of the number of detectors at any given time, and $S_t.IP_i$ is 1 if and only if a detector is located in the ith host at time t. Function *isDetectBot(sig)* decides if the bot pattern changes judged from the signature of bot traffic, *minDetectionProb()* is the function to calculate the lower bound of the probability of detecting bot traffic, given the current detector locations, and *attackUncertainty()* is the function to measure the uncertainty created against the bots with respect to the location of the detectors. The details of the functions can be found in [27].

4 MTDSYNTH ARCHITECTURE

The MTDSynth framework provides 1) an agility language specification HAPL to create agility control programs (agility policies for MTD and cyber deception), 2) an open programming environment to develop rapidly and safely sense-making and decision making MTD actions to enable cyber agility and dynamic cyber defense capabilities on Software Defined Networking using OpenDaylight[21] controller. Figure 3 show's the architecture of MTDSynth. The MTDSynth framework can be divided into three main components: 1) MTD policy specification interface, 2) MTD controller, and 3) ActiveSDN Engine. For scalability and rapid enhancement, MTDSynth designed as a plugin play model so that any part of the framework can be extended, modified or even replaced by adding other services as an application that will run in a separate process. For example, besides SMT solver, ASP[10] or ConfigChecker[6] can be used to solve constraints satisfiability problems. The communication between all components in MTDSynth occurs through REST API using JSON objects.

4.1 Interface

The MTDSynth framework provides an interface for MTD policy creation leveraging the agility language specification in figure 2. Authentic users can use the interface to generate agility policies

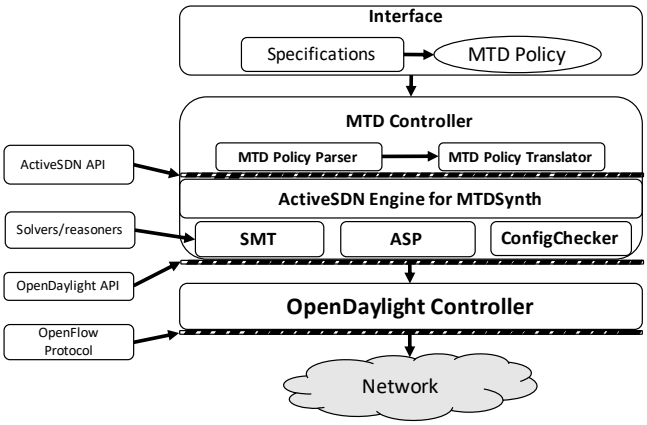


Figure 3: MTD Controller Synthesis using MTDSynth.

for rapid deployment of MTD defense or deception with full concurrency and safety. The policy creation interface provides an easy description of how to follow the agility language specification to create the correct MTD policy.

4.2 MTD Controller

The MTD controller in MTDSynth framework is the main orchestrator that handles the end-to-end processing from MTD policy to safe deployment of that policy in the network. It has three modules: a parser, a policy translator and an engine.

MTD policy parser. The policy parser is the first step to process the MTD policy created by the interface. It parses the given policy according to the language specification, generates a parsing tree and checks any syntax error. Then the parse tree is delivered to the translator. If any error occurs while processing the policy, the parser provides feedbacks to the interface; therefore, the user can make necessary correction in the policy. The MTD policy parser also measures the semantic correctness of the given policy from the parse tree. It checks whether the policy uses any primitives, sensors, mutation functions, or constraints that are not supported by ActiveSDN. Moreover, it checks the correct use of arguments in different primitives, type mismatch while comparing the output of different primitive constrains with each other, etc. If the validation fails, the reason behind the failure is returned to the interface as feedback.

MTD policy translator. The translator module does the primary task for the MTD controller, translate the MTD policy to a python script with the ActiveSDN API. The MTDSynth framework expose the primitives to create MTD policy from all the actions, sensors, and constraints ActiveSDN engine has in the ActiveSDN API. The translator generates a complete python script from the given policy based on that API, therefore executing the script will deploy the policy into the network through ActiveSDN engine. The translator knows how to invoke each function the engine provides correctly and by correct, it means the proper argument selection, providing appropriate response to any notification, etc. The script can run as a daemon server to communicate back and forth with the engine. The reason behind that, some API will generate output that can be used as input to another API (e.g., the output of *getCriticalLink(), l* can be used as input in *isLinkFlooding(l, ...)*). Thankfully, ActiveSDN engine is a multithreaded module that can handle multiple request at a time from the translator program, which improves the efficiency of MTDSynth framework in MTD policy deployment. In section 5 we describe how ActiveSDN API used for MTD policy deployment in MTDSynth framework through several examples.

ActiveSDN engine. The ActiveSDN engine in MTDSynth framework is built on top of an OpenDaylight SDN controller that provides the main programmable environment for generating safe MTD policies. It leverages the SDN capability for rapid deployment of critical MTD functions such as mutation or deception automatically orchestrating low-level network configuration changes. ActiveSDN engine enhances the automation process by introducing a rich set of primitives, the ActiveSDN API, which can be used directly for immediate deployment of defense and deception actions. The engine has solvers like SMT, ASP, or other third-party solutions like *ConfigCheker* to verify or resolve complex constraints. These modules run as a standalone service in different process side by side with the engine. The engine uses the OpenFlow protocol[20] to communicate with the SDN network. Appendix B shows the API list; however, the interested reader is referred to [14] for a complete API list with a proper description of each primitive.

4.3 Workflow

ActiveSDN implements the MTD actions primitives and publishes them into ActiveSDN API. The MTD controller in MTDSynth configures these actions automatically based on the parameter and constrains user enforces while creating MTD policies. Then using the API, the MTD controller deploy the actions into the network. To solve the constraints, the controller uses constraint primitives, sensor or solvers from the engine. All these communications happen using JSON object through REST API.

4.4 MTD Controller Synthesis

MTDSynth converts the agility specification to state diagrams, and generate the satisfiable MTD parameters and configurations through synthesis. Thus, we can realize an agility specification as a state transition where each state represents a mutation action and the transitions are based on environment actions including adversary, configuration or time-based events. We define the MTD parameters and the associated configuration changes as an MTD scenario. The MTD scenario should be guaranteed to achieve its goal despite any changes in the environment actions. In addition, the high-level logic that governs the behavior of the MTD scenario needs to be properly integrated with the MTD controller that regulates the system configuration. Therefore, we can formalize an MTD synthesis controller problem under verifiable guarantees as follows: Agility Synthesis Controller Problem. Given the (1) Agility specifications for MTD techniques(parameters, actions, and constraints expressed in Linear Temporal Logic (LTL)), (2) Environment specifications (attack model, topology and system configurations), the task of the MTDSynth agility synthesizer and control system is

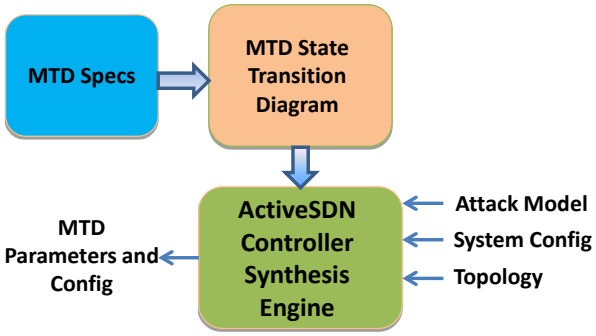


Figure 4: MTD Controller Synthesis

to generate a sequence of configuration control signals that, by construction, ensures that the system satisfies the model requirements for the MTD techniques. Figure 4 shows the framework of MTDSynth controller synthesis.

5 CASE STUDY

In this section, we will discuss the details about the safe deployment of MTD policies into the network through rigorous examples. We created a vast number of case studies to test MTDSynth for deploying all existing MTD techniques that are mostly used. Also, we created deception example using MTD techniques as well.

MTDSynth can do deception through spatial mutation to defend critical resources in the network within a second. The policy mentioned in section 3.3.2 for spatial mutation can create deception against scanning attack. In spatial mutation the IP address will be mutated based on each host pair in the network. That means, every host in the network will communicate with the list of mutable critical hosts through a different IP address. Meanwhile, any direct communication with the mutable host can be redirected to honeypot to deceive the attacker.

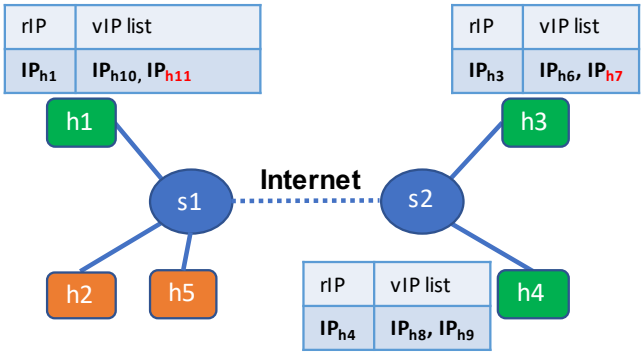


Figure 5: Spatial mutation example.

5.1 Spatial Mutation

Following the MTD rule in section 3.3.2, the spatial mutation is explained using a simple network having $n = 5$ hosts h_1, h_2..., h_5 and two switches s_1 and s_2 shown in figure 5. From these hosts, critical resources h_1, h_2 and h_3 (number of mutable hosts, $m = 3$) will be protected through spatial mutation, and the mutation IP space must not have a collision (≤ 0.1). That means mutable host list $<h_1, h_2, h_3>$ each having a distinct virtual IP addresses (vIP) for every other mutable hosts in the network. Which concludes, the vIP address space will contain $m \times (m-1) = 3 \times (3-1) = 6$ six distinct unused IP addresses (for example, h_6 *to* h_{11}), two for each mutable host. Table 1 shows the DNS records and table 2 shows the flow rules for the spatial mutation. The framework will dynamically configure the DNS record and flow rule modification in corresponding switches. The first record in table 1 means, h_1 will reach host h_3 and h_4 with vIP h_7 and h_8 respectively. When h_1 try to communicate with h_3, from the DNS record, it figures out the destination IP is IP_{h_7} for host h_3. Now, the flow rule one for switch s_1 in table 2 states that, when real host h_1 sends packets to host h_7, the flow will be delivered to real host h_3 by setting the destination IP IP_{h_7} to IP_{h_3}. When the packet arrives to the destination edge switch s_2, the rule two in switch s_2 changes the source IP from IP_{h_1} to $IP_{h_{11}}$. After receiving the packet, when destination host $h3$ looks in the DNS to resolve who is the sender IP $IP_{h_{11}}$, it founds host h_1 as the sender. Rule three in switch s_1 is for the reverse flow when real host h_3 reply to h_1, the source IP will be changed back to vIP IP_{h_7} from real IP IP_{h_3}. Therefore, h_1 will be transparent about the mutation and will assume that h_3 actually serves in vIP IP_{h_7}. Rule four describes the same way as rule one, when host h_3 communicates with host h_1. From the DNS record host h_3 obtains the IP for h_1 is $IP_{h_{11}}$. Therefore, when host h_3 sends a packet to $IP_{h_{11}}$, rule four changes the destination from $IP_{h_{11}}$ to IP_{h_1} and forwards. In appendix A, we showed two screenshots of the real flow rules in the edge switches while spatial mutation was enabled. Table 3 shows the API for spatial mutation.

h1→ (h3, **IP$_{h7}$**), (h4, **IP$_{h8}$**)
h3→ (h4, **IP$_{h9}$**), (h1, **IP$_{h11}$**)
h4→ (h3, **IP$_{h6}$**), (h1, **IP$_{h10}$**)

Table 1: DNS record for Spatial Mutation.

Flow	s1	Internet	s2	Flow
$IP_{h1} \rightarrow IP_{h7}$	❶src=IP_{h1},dst=IP_{h7};set_dst:IP_{h3}	$IP_{h1} \rightarrow IP_{h3}$	❷src=IP_{h1},dst=IP_{h3};set_src:IP_{h11}	$IP_{h11} \rightarrow IP_{h3}$
$IP_{h1} \leftarrow IP_{h7}$	❸src=IP_{h3},dst=IP_{h1};set_src:IP_{h7}	$IP_{h1} \leftarrow IP_{h3}$	❹src=IP_{h3},dst=IP_{h11};set_dst:IP_{h1}	$IP_{h11} \leftarrow IP_{h3}$

Table 2: Flow rules for Spatial Mutation.

Using spatial mutation, MTDSynth can create deception into the network. In the API list, there is a mutation action called *createShadow()* that enables deception while doing spatial IP mutation. Figure 6 shows, by using proxy P, MTDSynth initiates deception to protect host h_4 in a network where an adversary may compromise host h_1 and targeting to reach critical server h_7. To disrupt the adversary lateral movement, MTDSynth will create a shadow network ($h_{31} - h_{36}$) based on the *createShadow()* action to confuse the adversary by falsifying the overall network view. Moreover, MTDSynth will deploy a generic rule $(src=^*, dst=IP_{h_4}, set\_dst:IP_{proxy})$ in all

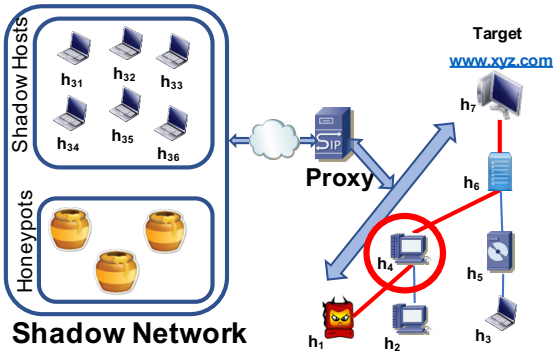


Figure 6: Deception by MTDSynth.

MTD Actions	Parameters	Descriptions
spatialMutation()	*<h>*	List of ***h*** mutable host.
	N	Number of total host in the network.
	<unused_range>	Unused IP address list. (Also can be provided with start address and range limit.)
	< m_i >	Mutable address per host = ***(n-1)*m_i***
	when	**-1**: Deactivate Special IP mutation.
		0: One time mutation.
		x: Time based mutation. Mutate IP after *x* seconds.
		other-mutation: For future use.
	how	**Uniform Distribution, Random Distribution**: Select mutable address list for each mutable host from ***<unused_range>***

Table 3: Spatial mutation API.

switches for any traffic destined to h_4 forward to *P*. Therefore, if adversary directly probes or try to reach h_4, her traffic will be redirected to *P*. In *P*, there are two rules, rule (1) *(src=*$IP_{adversary}$*, dst=*IP_{h_4}*)* $\rightarrow$ *(src=*IP_{proxy}*, dst=*$IP_{honeypot}$*)*. This will forward the traffic to the honeypot in the cloud to generate a response for adversary traffic. Rule (2) *(src=*$IP_{honeypot}$*, dst=*IP_{proxy}*)*$\rightarrow$*(src=*IP_{h4}*, dst=*$IP_{adversary}$*)* will send the reply traffic back to adversary deceiving that the real host h_4 is replying instead of honeypots. To avoid traffic congestion on *P*, several instance of P can be deployed on the network.

5.2 IP Mutation

IP mutation is an MTD action in MTDSynth. IP mutation mutates or changed the source and/or destination IP address to hide the actual/real IP address from the end-users, whereas the mutation itself is transparent to them. Besides, the mutation doesn't hamper any regular communication between hosts. The end host is unaware of the mutation. IP mutation helps to defend against any scanning attack for collecting network information for reconnaissances. Moreover, the scanner gets detected immediately.

The ipMutate function has three configurable parameters: the list of the *real* host IP address called *rIP* list that will get new mutated virtual IP address *vIP*, the virtual IP address space as *vIP* list and the mutation function *f*. For each *rIP* a *vIP* will be chossen from the *vIP* list using the mutation function *f*. For a periodic IP mutation, the event trigger need to be a time interval Δ, so that the *rIP* will be mutated every Δ seconds.

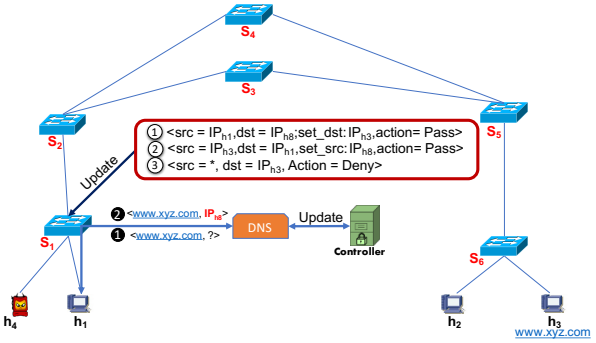


Figure 7: IP Mutation Example.

Figure 7 shows an example of IP Mutation, where h_1, h_2, h_3, h_4 are the the hosts and $s_1, s_2, ..., s_6$ are the switches. A web server *www.xyz.com* is running in h_3 and host h_4 is a scanner. Assume that, IP Mutation is activated with *rIP* as IP_{h_3}, mutation function f as a random function with a range $(1, 10)$ means the the *vIP* list contains a range of ten unused IP addresses (e.g., IP_{h_5} to $IP_{h_{14}}$). If the mutation time interval is Δ, then every Δ seconds, *rIP* IP_{h_3} will get a random *vIP* form the *vIP* list. When IP Mutation starts, in each mutation interval ActiveSDN installs three (the third rule is optional) flow rules into corresponding edge switches, switches that are near to the end hosts and update the DNS entry record. Assume that, for such an interval, the random *vIP* is chosen as IP_{h_8}, the corresponding rules are mentioned in figure 7.

Now, when host h_1 try to reach webserver *www.xyz.com*, it makes a DNS query to obtain the IP address of the webserver in step 1. The gateway switch s_1 forwards this request to DNS server. The DNS record is already get updated by the controller and instead of giving the *real* IP address IP_{h_3}, in step 2 the DNS server reply the *vIP* IP_{h_8} as the DNS query request. After achieving the IP address of the webserver, h_1 sends packets setting the destination IP address as IP_{h_8}, and flow rule (1) get matched in s_1. Rule (1) sets the destination IP from IP_{h_8} to IP_{h_3} and forward the packets. When the reply coming back from the webserver to h_1, rule (2) matched that changes back the source IP from IP_{h_3} to IP_{h_8}. Hence, h_1 and h_3 continue their communication withing being known to the mutation.

For scanner h_4, it does not go to the DNS server to obtain IP addresses for any host, rather try to probe by random IP address. When h_4 probe h_3 by IP IP_{h_3} directly, rule (3) matches, and the flow gets dropped. Rule (3) denies any communication that directly forwards to the destination IP address IP_{h_3}. Note that, this rule is optional in IP mutation.

5.3 Path Mutation

The path mutation is another MTD action that mutates the *data path* based on a flow between the source and the destination host without interrupting their regular communication. The existing path will be removed, and a new path will be installed depending on the mutation parameter. Path mutation is useful to overcome any link flooding attacks such as the Crossfire Attacks [17] because this technique confused the attacker to fix a specific critical link to flood. Moreover, path mutation provides proactive defense ability into the network restraining attacker permanently from link flooding.

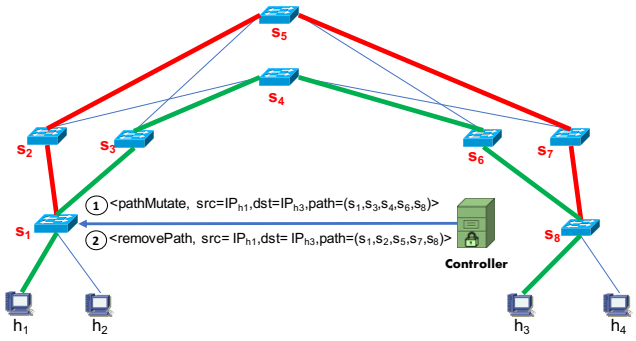


Figure 8: Path Mutation Example.

The path mutation API has two main parameters: the flow between a given source-destination IP addresses and a path profile. The path profile is the constraint user made while creating the MTD policy for path mutation. The path profile measures the overlap between current and new mutable path, maximum path length, minimum bandwidth, and the risk a path can have. Besides, any specific *switches* can be included or excluded in the new path. Solving all of these constraints, path profile will generate a mutable path space from where the next mutation path will be chosen. Finally, the event trigger defines how frequently the mutation will happen. If it's a time interval trigger, the path mutation will happen periodically.

Figure 8 is an example of path mutation. Before the mutation starts, source host h_1 communicates with destination host h_3, using path $(s_1, s_2, s_5, s_7, s_8)$. Then after path mutation gets triggered, MTD controler get a mutable path$(s_1, s_3, s_4, s_6, s_8)$ from path profile and configure *pathMutate* API using rule (1) $src = IP_{h_1}$, $dst = IP_{h_3}$ and path = $(s_1, s_3, s_4, s_6, s_8)$. This will install a new path for source h_1 to destination h_3. After mutation, instead of using the old path $(s_1, s_2, s_5, s_7, s_8)$, source h_1 now communicates with destination h_3 using the new mutated path $(s_1, s_3, s_4, s_6, s_8)$. This mutation is transparent to all end hosts while the communication in between them is uninterrupted.

Now the challenge in path mutation is to make uninterrupted communication between the end hosts using the new path instead of the old path. To do so we use *Priority* and *Timeouts* of the flow entries. According to OpenFlow specification [20] there are two fields available in the flow entries, 1) *Flow Priority*: matching precedence of the flow entry, if any packet matches with two different flow entries, the packet will follow the higher priority flow entry and 2) *Idle Timeout*: if it is set, then the flow entry will be expired (removed from the flow table) in the specified number of seconds if any packets are not hitting the entry.

To ensure the uninterrupted property of path mutation, there are two options: 1) explicitly call the *removePath* API that will delete the old path, or 2) install every flow rules in *pathMutation* setting the idle timeout with a specific number of seconds(usually the time interval of the event trigger). In MTDSynth, for each mutation cycle, to delete the previous path and install a new path, a new set of flow rules get deployed to the switches with a given idle timeout but higher flow priority from the previous flow rules. Therefore, if the packets find two sets of matching flow rules but different flow priorities, they follow the higher priority rules. The lower priority flow rules become idle and get removed from the flow table after *timeout* time, which means the old path get deleted.

6 IMPLEMENTATION AND EVALUATION

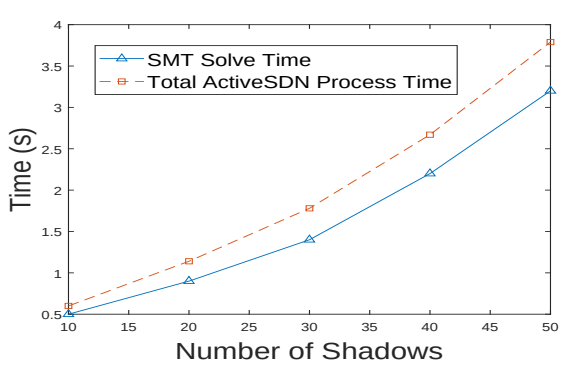
In this section we will describe how we developed MTDSynth framework and show the effectiveness through rigorous testing. We built ActiveSDN framework using OpenDaylight controller in Java and use ActiveSDN engine and ActiveSDN API to make the MTD policy syntheiszer: MTDSynth. We run MTDSynth in an iMac machine having 32GB of RAM and 4 GHz Intel Core i7 processor with macOS Mojave. We have a physical EdgeCore SDN Switch running Pica8R PicOS with virtual SDN Open vSwitch(v1.3) network running in different machines each having 32GB of RAM 8 CPUs in Ubuntu 16.04. We use Mininet[1] to create the virtual SDN network and Vagrant[3] for managing dynamic creation of shadow hosts in the network. We develop the MTD controller in python and run it as a different process alongside with the MTDSynth. The MTD controller provides web service for the Interface and communicates with the ActiveSDN engine controller through the REST ActiveSDN API(northbound). ActiveSDN controller use OpenDaylight to communicate with the network SDN switches following OpenFlow protocol.

We evaluated the performance of MTDSynth for temporal RHM, spatial RHM, and RRM. The most important factor for MTDSynth is the mutation processing delay, which include the SMT solve time and the configuration delay. Here the SMT solve time is unaffected by the ActiveSDN implementation, so we compare the total processing delay with the SMT solve time.

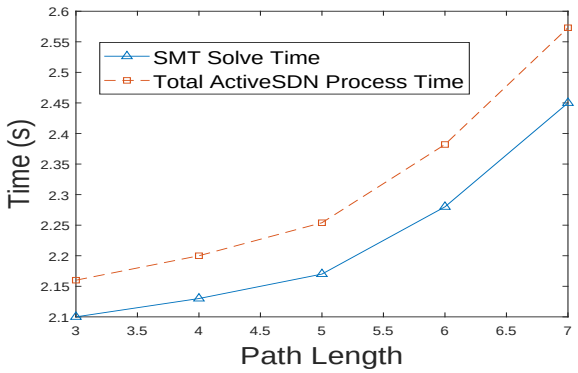
For temporal RHM, the controller needs to update the end-point IP and related DNS. We found that the average configuration delay is about 0.027s and this is unaffected by the network size. For spatial RHM, Figure 9a shows the total processing delay and the SMT solve time in a fixed network with 12 mutable hosts, with different number of shadow addresses. We can see that the difference between SMT solve time and total processing delay is small, which means the configuration delay is small (for example, about 0.24s for 20 shadows). For RRM, Figure 9b shows the total processing delay and the SMT solve time in a network of 200 hosts with a flow of fixed source and destination, and different required path length. We can also see that the configuration delay is small (for example, about 0.07s for path length 4).

7 CONCLUSION

In this paper we present a framework called MTDSynth to develop and deploy various MTD techniques fast and safely leveraging the open programming capability of SDN. We introduce a formal ontology and MTD language for agility. We show MTD language examples for temporal and spatial RHM, RRM, detector mutation. We build an open programming environment to develop rapidly and safely sense-making and decision making MTD actions to enable cyber agility and dynamic cyber defense capabilities in MTDSynth on top of ActiveSDN using OpenDaylight controller. In our implementation and evaluation we show that the average configuration delay for RHM and RRM is in the order of tens of milliseconds



(a) Spatial Mutation Performance



(b) Route Mutation Performance

Figure 9: ActiveSDN Performance

to hundreds of milliseconds, which validates the feasibility and scalability of our framework.

For future work, we want to integrate dynamic deception capability, adaptive intrusion response mechanism into the existing framework. We also want to add other classes of MTD techniques that are learning based, independent from configuration managements.

8 ACKNOWLEDGEMENT

This research was supported in part by the United States Army Research Office under contract number W911NF171043. Any opinions, findings, conclusions or recommendations stated in this material are those of the authors and do not necessarily reflect the views of the funding sources.

REFERENCES

[1] Mininet: An instant virtual network on your laptop (or other pc). http://mininet.org/.
[2] Software-defined networking. https://en.wikipedia.org/wiki/Software-defined_networking.
[3] Vagrant: Development environments made easy. https://www.vagrantup.com/.
[4] Ehab Al-Shaer. Mutable networks, National cyber leap year summit 2009 participants ideas report. Technical report, Networking and Information Technology Research and Development (NTIRD), August 2009.
[5] Ehab Al-Shaer. Toward network configuration randomization for moving target defense. In Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and X. Sean Wang, editors, *Moving Target Defense*, volume 54 of *Advances in Information Security*, pages 153–159. Springer New York, 2011.
[6] Ehab Al-Shaer and Mohammed Noraden Alsaleh. Configchecker: A tool for comprehensive security configuration analytics. In *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on*, pages 1–2. IEEE, 2011.
[7] Ehab Al-Shaer, Qi Duan, and Jafar Haadi Jafarian. Random host mutation for moving target defense. In *SecureComm*, volume 106, pages 310–327. Springer, 2012.
[8] Michael Atighetchi, Partha Pal, Franklin Webber, and Christopher Jones. Adaptive use of network-centric mechanisms in cyber-defense. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 183, Washington, DC, USA, 2003. IEEE Computer Society.
[9] Deborah Bodeau and Richard Graubart. Cyber resiliency engineering framework. *MTR110237, MITRECorporation*, 2011.
[10] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.
[11] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In *SBMF '09, Brazilian Symposium on Formal Methods*, 2009.
[12] Q. Duan, E. Al-Shaer, M. Islam, and H. Jafarian. Conceal: A strategy composition for resilient cyber deception-framework, metrics and deployment. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, May 2018.
[13] Qi Duan, Ehab Al-Shaer, and Jafar Haadi Jafarian. Efficient random route mutation considering flow and network constraints. In *CNS'13*, 2013.
[14] Md Mazharul Islam. Activesdn. https://github.com/rakeb/activesdn.
[15] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: Transparent moving target defense using software defined networking. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 127–132. ACM, 2012.
[16] Jafar Haadi H. Jafarian, Ehab Al-Shaer, and Qi Duan. Spatio-temporal address mutation for proactive cyber agility against sophisticated attackers. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 69–78, New York, NY, USA, 2014. ACM.
[17] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. The crossfire attack. In *2013 IEEE symposium on security and privacy*, pages 127–141. IEEE, 2013.
[18] Dorene Kewley, Russ Fink, John Lowry, and Mike Dean. Dynamic approaches to thwart adversary intelligence gathering. *DARPA Information Survivability Conference and Exposition*, 1:0176, 2001.
[19] Lisa M Marvel, Scott Brown, Iulian Neamtiu, Richard Harang, David Harman, and Brian Henz. A framework to evaluate cyber agility. In *MILCOM 2015-2015 IEEE Military Communications Conference*, pages 31–36. IEEE, 2015.
[20] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
[21] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*, pages 1–6. IEEE, 2014.
[22] J. Michalski, C. Price, E. Stanton, E. Lee, CHUA K. Seah, Yip Heng TAN, and C. Pheng. Final report for the network security mechanisms utilizing network address translation ldrd project. technical report sand2002-3613. Technical report, Sandia National Laboratories, 2002.
[23] John T. Michalski. Network security mechanisms utilising network address translation. *International Journal of Critical Infrastructures*, 2(1):10–49, 2006.
[24] Jose David Mireles, Eric Ficke, Jin-Hee Cho, Patrick Hurley, and Shouhuai Xu. Metrics towards measuring cyber agility. *IEEE Transactions on Information Forensics and Security*, 2019.
[25] R. Morehart. Evaluating the effectiveness of ip hopping via an address routing gateway. Master's thesis, AIR FORCE INSTITUTE OF TECHNOLOGY, 2013.
[26] M. Sifalakis, S. Schmid, and D. Hutchison. Network address hopping: a mechanism to enhance data protection for packet communications. In *IEEE International Conference on Communications, 2005. ICC 2005. 2005*, volume 3, pages 1518–1523 Vol. 3, May 2005.
[27] Sridhar Venkatesan, Massimiliano Albanese, George Cybenko, and Sushil Jajodia. A moving target defense approach to disrupting stealthy botnets. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, pages 37–46, New York, NY, USA, 2016. ACM.
[28] Zhenqiang Ye, Srikanth V. Krishnamurthy, and Satish K. Tripathi. A framework for reliable routing in mobile ad hoc networks. In *IEEE INFOCOM*, pages 270–280, 2003.

A APPENDIX: SPATIAL MUTATION FLOW RULES IN EDGE SWITCHES

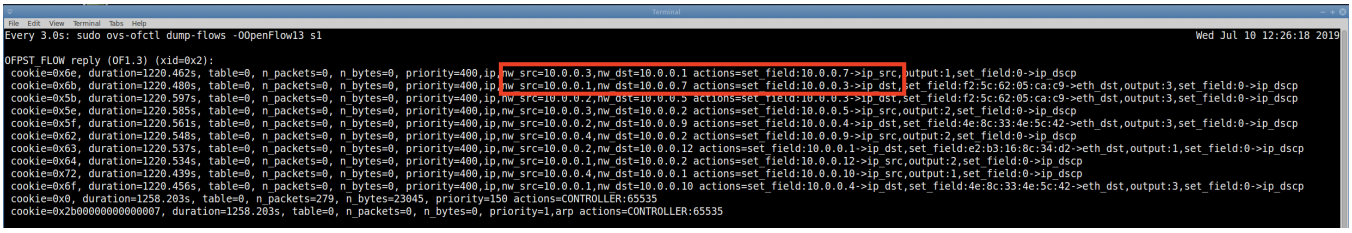


Figure 10: Real flow rules in source edge switch (s_1) for spatial mutation

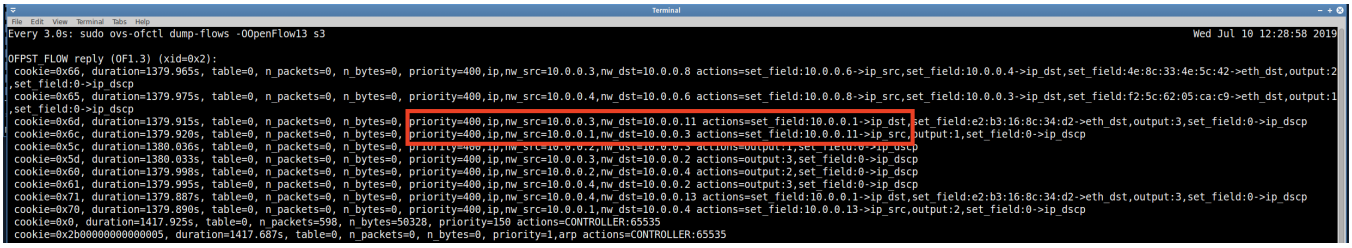


Figure 11: Real flow rules in destination edge switch (s_3) for spatial mutation

B APPENDIX: ACTIVESDN API

Type	Primitive	Descriptions	Output
Sensors	**isHostScanning(th, t)**	If any IP address sending SYN packets grater than the **threshold** *th* in a certain **time window** *t*, this function generates a true positive alarm.	boolean
	isLinkFlooding(l, th)	If the bandwidth consumed by the flows going to that **link** *l*, is greater than the the bandwidth **threshold** *th*, this will generate a true positive alarm.	boolean
	chekUDPICMPRate(f)	Calculate the average rate of a specific **flow** *f* of all UDP and ICMP flows.	<f>
	checkElephantTCP(<f>)	Calculate the percentage of large-size TCP traffic from a given **flow list** *<f>* .	<f>
	getFlowStatistics(f)	To get the complete information about a **flow** *f* such as: number of packets matched with *f*, bytes captured by *f*, time window for those packets, traffic type (ICMP, TCP, UDP) etc.	<flow stat>
	checkNewComers(<f>, t)	Calculate the ratio of new IP source addresses from a given **flow list** *<f>* that has not been seen before recently in a given **time window** *t*.	<f>
	getCriticalLinks()	This function returns the critical links may generated in the topology based on the flow data path.	<links>
	getAllFlowRules (s)	This function retrieves all the flow rules available into a **switch** *s*.	<f>
	findNeighbors(s)	Returns all the neighboring switches of the given **switch** *s*.	<s>
	findPortID(l, r)	It returns the port number of the **left switch** *l* that is connected to **right switch** *r*, if no link found, it returns -1.	numeric
	detectBot(sig)	If the signature of the examined packets satisfies the condition of bot traffic, return true.	boolean

Figure 12: ActiveSDN Sensors API

Type	Primitive	Descriptions	Output
Constraints	**isIncludeSwitch(R_t, <s>)**	Checks if **route** R_t contains all of the switches in the given **switch list** *<s>*.	boolean
	excludeSwitch(R_t, <s>)	Checks if **route** R_t doesn't contains all of the switches in the given **switch list** *<s>*.	boolean
	getRouteLength(R_t)	Return the number of links in the given **route** R_t	numeric
	getAvailableBandWidth(R_t	Checks the maximum bandwidth the given if **route** R_t has	numeric
	getFlowRate(s, d)	Calculates the number of packets transmitted in between **source** *s* and **destination** *d* per second	numeric
	overlap(R_t, R_{t+1})	Return the percentage of same links between two **route** R_t and R_{t+1}	number
	getRouteRisk(R_t)	Calculate the probabilistic risk that the given **route** R_t get attacked	number
	canReach(s, d)	Using ConfigChecker, find all reachable sources or destinations to/from a specific source *s* and destination *d*.	<route_list>
	checkUniqueIP(<ip>)	Checks whether all the elements in the given **IP list** *<ip>* is unique or not.	boolean
	checkNonRepeate (<ip1>, <ip2>)	Compare for all elements in the **IP list** *<ip1> and <ip2>* whether the *i*'th IP in *<ip1>* is different from the *i*'th IP in *<ip2>* . The length of these two list must be equal.	boolean
	checkSpatialCollision(<ip1>, <ip2>)	Find the collision probability that two distinct source using that same address to reach a destination.	numeric
	getMinDetectionProb(loc)	Calculate the lower bound of the probability of detecting bot traffic, given the current detector **location** *loc*.	numeric
	getAttackUncertainity(loc)	Measure the uncertainty created against the bots with respect to the **location** *loc* of the detectors.	numeric
	getAllPaths(s, d)	Calculates the all path **source** *s* and **destination** *d.*	<route_list>
	getShortestPath(s, d)	Calculates the shortest path **source** *s* and **destination** *d.*	<route_list>

Figure 13: ActiveSDN Constraints API

Type	Action Name	Descriptions
MTD Action	**ipMutate**	Change specific/all IP addresses of network host dynamically to specific/random IP addresses based on event, IP Mutation parameter and time. (This function is used for randomizing real src/dst IP addresses to virtual src/dst IP addresses so that real IP is used for routing but end Hosts always uses virtual IP addresses to communicate.)
	pathMutate	Change the path frequently of active flow(s) to another satisfiable path based on event or time.
	spatialMutation	Change the destination IP addresses based on the source, that is, different source may have different IP to reach the same destination.
	createShadow	Create a decoy dynamically for host *anonymity* and *diversity* based on CONCEAL framework [MTD16, CNS 18].
	reDirect	This function change the IP destination to redirect to a decoy, or tunnel the packet to a proxy then send back to its original destination.
	migrateService	Change the path of active flow(s) between **src** and **dst** including service migration of **dst** host based on new path and new **dst** host address.

Figure 14: ActiveSDN MTD Actions API