# Using Online Planning and Acting to Recover from Cyberattacks on Software-defined Networks

**Sunandita Patra,**[1,*] **Alexander Velazquez,**[2,*] **Myong Kang,**[2] **Dana Nau**[1]

[1]Dept. of Computer Science and Institute for Systems Research, University of Maryland, College Park, MD, USA
[2]Naval Research Laboratory, Information Technology Division, Washington, DC, USA
patras@umd.edu, alexander.velazquez@nrl.navy.mil, myong.kang@nrl.navy.mil, nau@umd.edu

## Abstract

We describe ACR-SDN, a system to monitor, diagnose, and quickly respond to attacks or failures that may occur in software-defined networks (SDNs). An integral part of ACR-SDN is its use of RAE+UPOM, an automated acting and planning engine that uses hierarchical refinement. To advise ACR-SDN on how to recover a target system from faults and attacks, RAE+UPOM uses attack recovery procedures written as hierarchical operational models. Our experimental results show that the use of refinement planning in ACR-SDN is successful in recovering SDNs from attacks with respect to five performance metrics: estimated time for recovery, efficiency, retry ratio, success ratio, and costEffectiveness.

## Introduction

*Software-defined networking* is a relatively new network management approach that enables dynamic, modular, programmatically efficient network configuration in order to improve network performance and simplify monitoring. Network management architectures generally have two layers: the data layer, where traffic flows and network packets are forwarded, and the control layer, which manages packet routing. In traditional network architectures, these two layers are highly coupled and the control is decentralized, which can lead to complexity and lack of agility. SDN architectures decouple the two layers and have a centralized control layer, implemented using a set of controllers.

SDNs are susceptible to a wide variety of known and unknown cyberattacks. With adversaries that can generate automated attacks at high pace and volume, as well as the possibility of system failures that can crop up at any time, it can be difficult for human system managers to perform the necessary recovery and defense tasks quickly enough.

In this paper, we introduce ACR-SDN, a system for recovering from cyberattacks and anomalies to a SDN. ACR-SDN is implemented as a management plane, which interacts with SDN components in the control plane and data plane. ACR-SDN has a Security Manager in charge of detecting anomalies within the SDN. Once an anomaly has been detected, ACR-SDN's actions for recovery are guided by RAE+UPOM, an online planning and acting engine based on hierarchical refinement (Patra et al. 2020b).

For each task that one might want RAE+UPOM to accomplish, a human expert may write various *refinement methods*, each of which is an alternative procedure for accomplishing the task. If the task is to recover from a particular kind of attack on a SDN, then each refinement method might specify a different way to try to recover. As cyberattacks become sophisticated, cyber responses have to be made sophisticated, and refinement methods are a good way to capture complex cybersecurity domain knowledge since they support all of the usual programming constructs, such as if-else statements, loops, and so on.

RAE+UPOM consists of two subsystems: RAE, an *actor* that executes refinement methods; and UPOM, a *planner* that predicts how well each refinement method is likely to work in the current situation. Given a task to perform, RAE will call UPOM to get advice on which refinement method to use. During RAE's execution of the recommended method, at each point where the method specifies another task to perform, RAE will again call UPOM to get advice on what method to use for that task. UPOM generates its advice by performing Monte Carlo rollouts, each rollout being a simulated execution of RAE in the current situation. The larger the number of rollouts, the better UPOM's advice will be.

Automated attack detection and diagnosis are not the focus of this paper. Rather, our focus is on recovery of the network from abnormal or insecure states. We assume that cyberattacks and system faults manifest as changes to the state. In our experimental evaluation, we generate these states directly, and then try to recover from them. In our architecture, anomaly detection is handled by the Security Manager, which sends tasks to RAE+UPOM, and additional diagnosis can be done via probing actions inside refinement methods.

This paper is organized as follows. This section is followed by related work. Then, we describe an example SDN attack-and-recovery scenario. We give some background on refinement acting and planning (RAE and UPOM). We describe ACR-SDN and how RAE+UPOM is integrated with it. The penultimate section gives experimental results, and the last section provides a concluding discussion.

[*]Contributed equally.

## Related Work

In today's world, cyberattacks are dynamic, fast-paced, and high-volume, while cyber responses are initiated at human speed. In current cyber defense systems (Zargar, Joshi, and Tipper 2013; Zhang et al. 2011; Mendes, Aloi, and Pimenta 2019; Yamaguchi 2020), most system adaptation and recovery processes are ad-hoc, manual, and slow, so keeping pace with existing and emerging cybersecurity threats is a challenging task. It is important to construct resilient computer systems that can autonomously protect and recover from cyberattacks and system failures.

Some effort has been invested to apply AI and ML techniques to SDN management and cybersecurity, but the field is still in its infancy. AI planning has been applied to manage SDNs (Gironza-Ceron et al. 2017), self-healing of SDNs has been studied (Ochoa-Aday, Cervelló-Pastor, and Fernández-Fernández 2019; Thorat et al. 2015), and RL has been applied to SDN flow rule management (Mu et al. 2018). Applying deep learning to security mainly focuses on the detection of intrusions and malware (KP, Alazab et al. 2020; Berman et al. 2019). One disadvantage of using supervised learning approaches is that they require a lot of training data, which can be difficult to obtain for SDN attacks. Recently, RL has been also applied for autonomous defense of SDNs (Han et al. 2018). However, (Han et al. 2018) did not consider typical cyberattacks to SDNs (see (Lee et al. 2020) for a summary of known SDN attacks) and used only simple actions (e.g., "isolate and patch a node"; "reconnect a node and its links"; "migrate the critical server and select the destination").

Attack detection is a topic that has been addressed many times. Various schemes exist to detect cyberattacks of different types (e.g., distributed denial of service (DDoS) attacks in SDNs (Lawal and Nuray 2018)) and network traffic anomalies (Bhuyan, Bhattacharyya, and Kalita 2014). However, little research has been done on autonomous cyber responses. In this paper, we do not deal with the detection of cyberattacks against a SDN, but rather we assume that they can be detected and focus on techniques for planning and executing autonomous responses.

Other systems exist that aim to protect information systems from cyberattacks. For example, (Shrobe et al. 2007) proposes a system that can be applied to existing software to provide attack detection and recovery. While we have a similar motivation in this paper, our work is more suitable for a distributed system like SDN (rather than a self-contained software application) and we take a different approach (refinement planning) to produce a target system with self-securing properties.

We are not aware of any existing approaches that use refinement planning for autonomous responses in SDNs.

Some other areas that have not yet been seriously considered in the existing literature are how the dynamic nature of SDNs, or IT systems in general, affect autonomous cyber responses and the extent to which complex expert knowledge may have to be utilized to deal with cyberattacks.

RAE (Ghallab, Nau, and Traverso 2016; Patra et al. 2020b) is based on an earlier system called PRS (Ingrand et al. 1996), which executed refinement methods somewhat like RAE's but did not consult a planner for advice on which method to use. PRS was extended with some planning capabilities in PropicePlan (Despouys and Ingrand 1999), a planner that used state-space search techniques.

The UPOM planner (Patra et al. 2020b) can reason about several aspects of real-world planning that PropicePlan could not handle, e.g., probabilistic outcomes of actions, exogenous events, and partial (rather than full) observability of the environment. UPOM's Monte Carlo rollout technique is based on the one used in UCT (Kocsis and Szepesvári 2006), an algorithm for decision-making on MDPs and game trees. However, UCT's search space is simpler than UPOM's because UCT has nothing like UPOM's refinement methods, and depends instead on searching over sequences of actions. We discuss this further in the next to next section.

## Example SDN Recovery Scenario

SDNs are vulnerable to various kinds of attacks and failures. Several SDN-specific attacks are discussed in (Yoon et al. 2017; Lee et al. 2020). Avenues of attack typically arise from weaknesses or vulnerabilities in SDN protocols (e.g., OpenFlow), software bugs (especially in SDN controller software, e.g., Floodlight, ONOS, OpenDaylight), and lack of authentication or encryption between components.

For any attack on a SDN, we can expect the symptoms to manifest as a change in system state. For example, in a *switch malfunction*, a switch exhibits unexpected behavior due to an internal error or some attack from outside, and in a *controller malfunction*, the controller's resource consumption goes out of bounds.

Throughout this paper, we use PACKET_IN flooding as an example of how a SDN can be attacked, how an attack can manifest as various symptoms throughout the system, and how ACR-SDN can recover the SDN through refinement acting and planning. This example also highlights the dynamic nature of the environment of an SDN. In a PACKET_IN flooding attack (see Figure 1), one or more malicious hosts continuously send traffic to an unknown (and possibly randomized) destination address.

In detecting such an attack, we first see CPU usage in the controller spike and remain high. One or more switches may report the controller as unresponsive, and they may experience increased CPU usage themselves. Some additional clues can include: *(i)* increased controller host table size, *(ii)* control plane network bandwidth saturation, *(iii)* increased control plane network latency, and *(iv)* increased CPU usage on the malicious host's switch.

To mitigate the attack, the malicious hosts could be rate-limited or disconnected from the network. However, it may take some time for the existing symptoms throughout the SDN to subside. This could be accelerated by clearing the controller host table (e.g., if it is tracking many thousands of bogus host addresses). If the controller remains unresponsive, it may be necessary to reinstall or replace it. Finally, if there are any especially critical data flows that have been affected, it could be beneficial to allocate a new switch and migrate critical hosts away from any over-burdened switches. In the next section, we discuss how such recovery and mit-

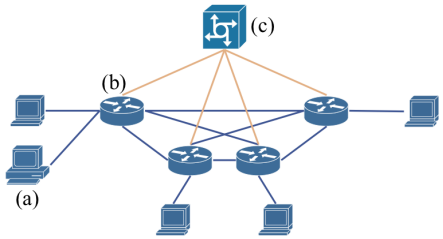



Figure 1: An example SDN with five hosts, four switches, and a controller. In a PACKET_IN flooding attack, host (a) sends many packets with unknown/randomized destination MAC addresses. When they are received at switch (b), they do not match any existing flow, so the switch forwards them to the controller (c) as an OpenFlow PACKET_IN request. Since the destination is unknown, the controller instructs the switch to flood the packet to all output ports. If the packet rate is high enough, and/or continues for long enough, the controller becomes overwhelmed and this can lead to denial of service of the whole network.

igation techniques can be implemented as refinement methods.

## Refinement Acting and Planning

This section gives an overview of RAE and UPOM. For further information about them, see (Patra et al. 2020b).

**Actor.** RAE (Refinement Acting Engine) is a system for performing tasks and responding to events in dynamic, unpredictable, and partially observable environments. In order to accomplish this, RAE takes as input a set of *refinement methods*, which are computer programs giving alternative ways of performing *tasks* or responding to *events*. A refinement method has the form:

method-name($arg_1$, $arg_2$, ..., $arg_k$)
task: task or event identifier
pre: test
body: program

A refinement method for a task or event $t$ specifies *how* to perform $t$, i.e., it gives a procedure for accomplishing $t$ by performing sub-tasks, commands, and state variable assignments. The procedure may include any of the usual programming constructs: if-else statements, loops, etc.

For example, consider the PACKET_IN flooding described in the third section. There are several alternative methods for responding this event. One of them, m3_ctrl_mitigate_pktinflood($id$) (see Figure 2), searches for switches which have been marked as unhealthy by the ACR-SDN's Security Manager, moves all critical hosts away from each such switch to a newly added switch before attempting to fix the old switch, and finally clears the host table in the controller. A refinement tree using m3_ctrl_mitigate_pktinflood($id$) and with one unhealthy switch $s_1$ is shown at the bottom of Figure 2.

Formally, RAE models a domain as a tuple $\Sigma = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$ where,

- $S$ is the set of states (e.g., states of the SDN);

```
m3_ctrl_mitigate_pktinflood(id)
  event: packetIn-flooding(id)
  body:
      if is_component_type(id) ≠ 'CTRL': fail
      # Detect which switches are the source of attack
      for s_id in state.components:
          if is_component_type(s_id, 'SWITCH')
          and not is_component_healthy(s_id):
              # Move critical hosts away
              if is_component_critical(s_id):
                  add_switch (s_id) # Add new switch
                  # Move critical hosts from bad switches
                  move_critical_hosts(s_id, s_id + '-new')
              fix_switch(s_id) # Fix unhealthy switch
      # Clear controller state
      clear_ctrl_state_besteffort(id)
      # Check whether controller is now healthy
      if not is_component_healthy(id): fail
```

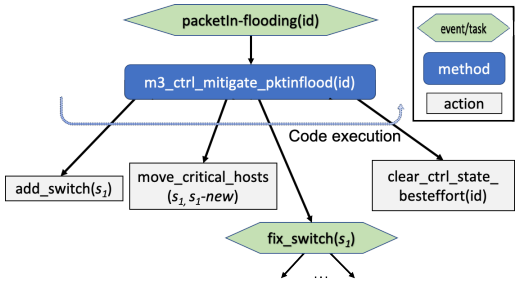



Figure 2: A refinement method and a partial refinement tree

- $\mathcal{T}$ is the set of events (e.g., attacks) and tasks (e.g., recovery from attacks) that ACRS may have to deal with;
- $\mathcal{M}$ is the set of methods for handling tasks or events in $\mathcal{T}$;
- $\mathcal{A}$ is the set of primitive actions (e.g., commands that can be executed on the SDN).

**Planner.** For each task or event in $\mathcal{T}$, $\mathcal{M}$ may contain several refinement methods, each describing a different way to perform the task or respond to the event. Which of these methods is best to use may depend on the specific situation. RAE can be configured to run purely reactively, in which case it will make this choice arbitrarily. RAE can also be configured to call a planner each time it needs to make a choice, so that the choice will be informed by the planner's predictions of each refinement method's potential outcomes. In the RAE+UPOM system, the planner that RAE uses is UPOM (UCT-like Procedure for Operational Models).

AI planning systems typically represent actions using *descriptive models* written in a language such as PDDL (McDermott et al. 1998; Haslum et al. 2019). These tell what the action will do, but not how to perform the action. In contrast, UPOM plans using the same refinement methods that RAE uses, doing simulated execution of the methods, as we will describe shortly. This gets rid of several issues that may arise when the models used for planning and acting are inconsistent, such as plan verification and plan management.

**Acting and planning.** The deliberative acting problem for ACR-SDN can be stated informally as follows: given $\Sigma$ and

a recovery task or event (an attack to the SDN) $\tau \in \mathcal{T}$, what is the "best" method $m \in \mathcal{M}$ to accomplish (or recover from) $\tau$ in a current state $s$? For the example of a PACKET_IN flooding attack, this reduces to choosing one among the three possible candidates, one of which is the m3_ctrl_mitigate_pktinflood($id$) method in Figure 2. ACR-SDN requires an online selection procedure which designates for each task or sub-task at hand the best method for pursuing attack recovery in the current context.

The current context for an incoming attack $\tau_0$ is represented via a *refinement stack* $\sigma$, which keeps track of how much further RAE has progressed in recovering from $\tau_0$. The refinement stack is a LIFO list of tuples $\sigma = \langle(\tau, m, i), \ldots, (\tau_0, m_0, i_0)\rangle$, where $\tau$ is the deepest current sub-task in the refinement of $\tau_0$, $m$ is the method used to recover from $\tau$, $i$ is the current instruction in $body(m)$, $\sigma$ is handled with the usual stack push, pop and top functions.

When RAE addresses a task or event $\tau$, it must choose a method $m$ to handle $\tau$. Purely reactive RAE makes this choice arbitrarily; more informed RAE relies on a planner. Once $m$ is chosen, RAE progresses on performing the body of $m$, starting with its first step. If the current step $m[i]$ is a primitive already being executed on the SDN, then the execution status of this action is checked. If the action $m[i]$ is still running, stack $\sigma$ has to wait, RAE goes on for other pending recovery tasks, if any. If action $m[i]$ fails, RAE examines alternative methods for the current sub-task via a procedure called Retry. Otherwise, if the action $m[i]$ is completed successfully, RAE proceeds with the next step in method $m$.

Planning with UPOM searches through this space by doing simulated sampling of the action's outcomes from a probability distribution decided by a human expert. UPOM (UCT-like Procedure for Operational Models) performs a recursive search to find a method $m$ for a task $\tau$ and a state $s$ approximately optimal for a utility function $U$. It is a UCT-like (Kocsis and Szepesvári 2006) Monte Carlo tree search (MCTS) procedure over the space of refinement trees for $\tau$. The mapping from UPOM's search to a UCT search is rather complicated; for details see (Patra et al. 2020b).

**Convergence of the planner.** UPOM converges when there are no infinite paths in the search space (for a proof, see Appendix A of (Patra et al. 2020a)). For scenarios that may contain infinite paths, the proof generalizes as follows. UPOM can be run using a limited depth $d$, in which case each rollout continues until it reaches depth $d$ or terminates early. At depth $d$, UPOM estimates the remaining utility using a heuristic function. It is straightforward to prove that for sufficiently large values of $d$, if the heuristic function is admissible then UPOM will still converge to the optimal choice. The basic idea is that if $d$ is large enough to traverse a large portion of the infinite path, then UPOM will get a very high estimated cost for any rollout that pursues the infinite path.

The planning of UPOM is not myopic because irrespective of where it is in the refinement tree, it searches for the next *best* action for the root task $\tau$ (not the current sub-task being refined). At any stage, the progress the actor, RAE, has made towards accomplishing the root task $\tau$ is captured by the refinement stack. A rollout continues until the refinement stack becomes empty, i.e., $\tau$ has succeeded or failed. The utility of the rollout is updated accordingly. After all the rollouts are done, UPOM chooses a refinement method $m$ for any sub-task of $\tau$ based on its estimated utility value for $\tau$.

## The ACR-SDN System

**Architecture.** The architecture for ACR-SDN consists of a number of components, organized into three layers: (1) the *Presentation Layer* provides a GUI to keep humans in the loop with respect to the operation of the system; (2) the *Security Layer* contains components that gather situational awareness and provide security services (e.g., monitoring and diagnosis) for the rest of the system; (3) the *Intelligent Planning Layer* deals with decision-making and is responsible for planning courses of action to be executed on the SDN. These components all are in the SDN management plane and interact with the SDN components (switches, controllers) (see Figure 3).

Figure 4 illustrates the components and data flows necessary for our implementation of the ACR-SDN architecture.

- The *Management Layer* is in the SDN management plane.
- The *Infrastructure Layer* is the underlying platform responsible for provisioning the virtual machines that act as our SDN controllers and switches.
- The *Control Layer* is in the SDN control plane and allows for monitoring of SDN components and executing actions in the SDN controllers and switches.

There are many different flavors and implementations of SDNs. In this paper, we utilize a software-based SDN as that target system that we defend, where the switch and controller software processes run inside virtual machines (VMs). The VMs serve as useful building blocks and allow us to have more diverse actions in our operational model (e.g., restart VM, add VCPU). With slight modifications to the operational model and action space, our approach could be adapted to defend other types of SDNs, including those with hardware-based switches.

We leverage the Web Application Messaging Protocol (WAMP, see https://wamp-proto.org/) to provide one-to-many and one-to-one communication between the various ACR-SDN components via publish-subscribe messaging (Pub-Sub) and remote procedure calls (RPC).

A central WAMP router in the management layer allows the State Manager, Security Manager, and WAMP Relays to communicate. The State Manager persists state information in a database. This database is accessed on-demand by the Web Server to provide situational awareness to human operators via the GUI. The WAMP Relays allow for controlled access to components in the SDN (control layer) and virtual machine platform (infrastructure layer), which allows system state monitoring and the execution of probing actions and corrective actions. Ultimately, monitoring and execution is handled by various agents in the infrastructure and control layers: LibVirt, SDNCTL, and SysMon.

The Security Manager receives system statistics, log messages, and alerts from the SDN components (switches, controllers). It monitors the health of each component and the

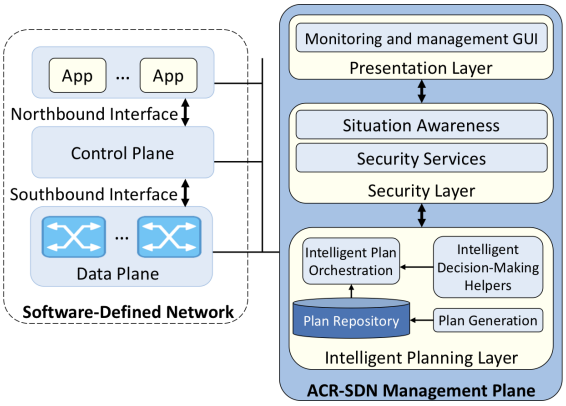



Figure 3: ACR-SDN architecture

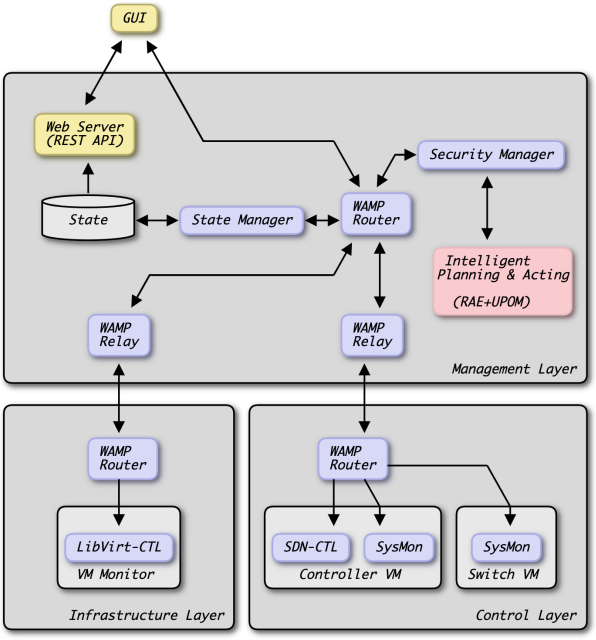



Figure 4: Data flows among components in ACR-SDN

overall health of the network, and uses configurable thresholds to determine when a task needs to be submitted to the planner. It also acts as the execution platform for RAE; when it receives a command, it executes it on the SDN and returns the status back to RAE.

RAE, the actor, is integrated directly with the Security Manager as a Python module, and they communicate with each other asynchronously using a set of shared queues.

## Integration of ACR-SDN and RAE+UPOM

The implementation of ACR-SDN in the SDN management plane has a security layer and an intelligent planning layer (Figure 3). The ACR-SDN Security Manager, Refinement Acting Engine (RAE), and the planner (UPOM) interact with each other and work together to defend against, and recover from, attacks to a SDN. A defense system for SDNs continuously monitors the switches and controllers and ensures that the target system is behaving as expected. To accomplish this, it must:

**Step 1.** Detect that an attack has occurred on the SDN.

**Step 2.** Diagnose what kind of attack has occurred. An attack may be detected when the network reaches some inconsistent state, or shows irregular behavior.

**Step 3.** Come up with an online plan to recover from the damage this attack has caused. The recovery process differ depending on current state of the system and the nature of the attack. The planner may re-plan when necessary. The Security Manager may or may not choose to use the planner, UPOM, via configuration parameters of RAE.

**State definition for SDN.** The state consists of two top-level dictionaries: components and stats. In order to secure a SDN, ACR-SDN has two types of components that it must deal with: controllers and switches. The components dictionary maps from component IDs to a nested dictionary containing keys that map to properties of the component (id, type, critical, etc.). The "critical" property warrants some explanation. It is a boolean flag that denotes whether a component is currently serving a mission-critical purpose, from a system requirements perspective. There may be many data flows passing through the network at any given time, but not all are critical for the system to meet its requirements. A switch might serve some combination of critical and non-critical data flows. This property allows for context-sensitive planning (e.g., move all critical hosts from one switch to a new switch, in order to temporarily insulate them from an ongoing cyberattack).

The stats dictionary maps from component IDs to a nested dictionary containing the keys health, cpu_perc_ewma, and potentially a number of other keys for statistics that may depend on the component's type (e.g., a switch will have flow_table_size). Each of these maps to another dictionary containing the key's value, the current value of this statistic, and thresh_exceeded_fn, a function that evaluates to true if the value exceeds the configured threshold (a numeric value in valid range of the state variable chosen by a human expert) for that statistic.

The state representation is designed to accommodate the dynamic nature of the SDN. The number of components (switches and controllers) can change at any time for various reasons: an unexpected failure or outage, a course of action that includes adding or removing a component, performing moving target defense, etc. As components are added or removed, the top-level dictionaries are simply updated to reflect the new set of components. This goes hand-in-hand with the operational model, which is designed with this state representation in mind and enables effective planning no matter what the current system configuration looks like.

**Communication between components.** The communication between the Security Manager and RAE takes place using three shared queues:

- **Task queue**: After the Security Manager detects an attack, it puts an attack event or a recovery task on the task queue. The task stays in the queue until RAE reads from it and

chooses an applicable refinement method. RAE calls the planner, UPOM for making this choice.

- **Command execution queue**: After planning using UPOM, RAE sends commands (atomic actions to be executed) to the Security Manager by putting them in the command execution queue one by one. The Security Manager reads the command from the queue.
- **Command status queue**: After executing a command, Security Manager puts the information about whether a command succeeded or failed and next state of the SDN in the command status queue. RAE reads this information and updates the state accordingly.

Within a refinement method, if more information is needed than is currently available in the state, then a probing action is used to request this information from the Security Manager. For example, if a switch component is misbehaving and the course of action depends on whether its flow table is over-filled, but the size of its flow table was not available in the state, then the get_switch_flowtable_size action can be requested. When this command returns successfully, then flow of control can continue and the missing value will have been included in the updated state.

The Security Manager has a process that continually checks the execution queue. When a command is available, it reads the command name and parameters from the queue. It then looks up the command by name and dispatches it as appropriate. Some commands need to run on the component itself, while others need to run on the underlying platform (e.g., the virtual machine monitor) that the component is running on. In either case, the Security Manager notifies RAE of success or failure (along with a copy of the updated state) via the command status queue.

**Action and environmental model.** Each low-level action is modeled in the operational model and has a counterpart on the execution platform (the Security Manager), so that when an action is passed by RAE to the Security Manager, it can be carried out on the SDN. These actions have costs associated with them (defined by experts in the operational model), which are estimates of how long it will take to implement the action on the target system. While these estimates may not be absolutely accurate (e.g., they can vary depending on the underlying hardware specs of the target system), they should at least reflect reality when they are compared relative to each other. For example: restarting a virtual machine takes longer than adding or removing a flow rule; reinstalling controller software takes longer than clearing the controller's state. We arrived at our cost estimates using experimentation and analysis from domain experts.

When declaring an action in the operational model, we assign the name, any parameters (e.g., component ID), and code that modifies the state and return success or failure, to model the effect that the action is expected to have on the system. Preconditions are encoded in the operational model (either in a action function or refinement method) by checking the state and returning failure if the action does not apply to the current state. The actions are nondeterministic and the predictive models used by UPOM sample their outcome from a probability distribution. In the environment definition, the domain expert assigns an estimated probability to each action. Other strategies, such as learning from history or running the action in an emulated system, may also be used to guess how likely an action is to succeed.

## Experimental Evaluation

To test our SDN attack recovery system (ACR-SDN with RAE+UPOM), we modeled attacks to the SDN as events and component recovery tasks. We generated a test suite that represented three different classes of cyberattacks:

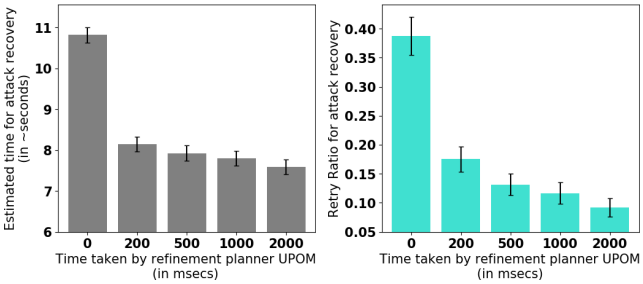
1. Attacks that exhaust controller memory, e.g., PACKET_IN flooding, switch table flooding, memory resource exhaustion (Yoon et al. 2017; Lee et al. 2020)
2. Attacks that exhaust switch memory, e.g., flow table flooding (Yoon et al. 2017; Lee et al. 2020)
3. Attacks that disconnect a switch from a controller, e.g., switch ID spoofing, malformed/corrupted OpenFlow message type (Yoon et al. 2017; Lee et al. 2020)

The test suite that we generated consists of 300 recovery tasks, 100 for each class of attack. Our operational model consists of 15 tasks, 22 refinement methods, and 16 commands. The individual tests were randomized in such a way that the initial assignment of state variables reflected symptoms that would be caused by the given class of cyberattack. The number of controllers in a given test ranged from one to four and the number of switches ranged from 16 to 64. One to three switches or controllers were randomly chosen to be attacked. We configured UPOM to optimize a linear combination of efficiency (reciprocal of estimated time, see (Patra et al. 2020b)) and probability of success. Each test was run 50 times to account for nondeterministic outcomes. We ran the tests on a simulated SDN running on a 2.8 GHz Intel Ivy Bridge processor.[1]

To measure ACR-SDN's performance, we measure five different metrics: the estimated time for attack recovery, the efficiency, retry-ratio, success-ratio (from (Patra et al. 2019)), and costEffectiveness (a linear combination of efficiency and probability of success). We discuss each of them as follows.

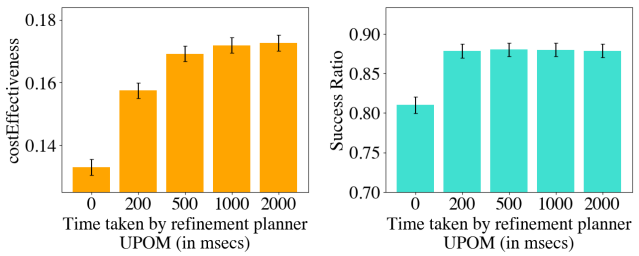
**Estimated time for attack recovery.** Figure 5(a) shows how the estimated time for attack recovery changes as we give more time to the refinement planner, UPOM. We observe that purely reactive RAE (with no planning, i.e., 0 time given to UPOM) is able to help the SDN recover from attacks in $\sim$11 seconds. Further, when doing refinement planning with UPOM, we observe $\sim$32% decrease in the estimated time for recovery. The error bars in the plot show 95% confidence intervals. Note that we used a Python implementation of RAE and UPOM which is a relatively slow programming language. Using faster programming language like C++ or Java could speed up the recovery process even further.

[1]Full code is at ⟨https://bitbucket.org/sunandita/rae/src/nrl-domain-airs/⟩.




(a) (b)

Figure 5: (a) Estimated time for attack recovery (in ∼seconds) and (b) retry ratio for attack recovery in ACR-SDN, with UPOM configured to optimize a linear combination of efficiency (reciprocal of estimated time) and probability of success. The error bars show 95% confidence.




(a) (b)

Figure 6: (a) Estimated costEffectiveness (a linear combination of reciprocal of estimated time and probability of success) for attack recovery and (b) success ratio for attack recovery in ACR-SDN, with UPOM configured to optimize costEffectivess. The error bars show 95% confidence.

**Retry ratio.** If a method for a task $\tau$ fails during execution, RAE looks at the list of untried methods for $\tau$ and chooses one among them. Each such choice is called a Retry. The higher the number of retries, the higher the execution time. Retry ratio measures the number of retries including all subtasks divided the total number of incoming tasks in RAE. Figures 5(b) shows the retry ratio for the experiments with the randomly generated test suite. From the plot, we can conclude that planning with a time limit of 200 msecs for UPOM outperforms purely reactive RAE with 95% confidence. Planning with a time limit of 2 seconds for UPOM outperforms planning with a time limit of 200 msecs with 95% confidence.

**Cost-effectiveness.** The cost-effectiveness is a linear combination of the reciprocal of the cost and the probability of success. Figure 6(a) shows that the cost-effectiveness improves gradually as UPOM is give more time to do a larger number of Monte Carlo rollouts. Using the planner, UPOM achieves close to 31% improvement in cost-effectiveness, compared to purely reactive acting.

**Success ratio.** Figure 6(b) shows how the success ratio (the number of attacks ACR-SDN is able to recover from successfully / total number of attacks) as we give more time to the refinement planner, UPOM. When doing refinement planning with UPOM, we observe close to 7% increase in the success-ratio.

In summary, we are able to automate attack recovery in SDNs using ACR-SDN and the refinement acting engine, RAE. Planning with UPOM further improves the performance in terms of estimated recovery time, efficiency, retry-ratio, costEffectiveness and success ratio with 95% confidence.

## Conclusions and Future Work

In this paper, we introduced ACR-SDN, a system for autonomous attack recovery in software-defined networks. ACR-SDN integrates RAE+UPOM into the management plane to help a SDN autonomously recover from failures and cyberattacks. Refinement methods for RAE+UPOM are recovery procedures written by human experts. A recovery procedure can be any complex algorithm with any programming constructs, such as if-else statements, loops, and so on. An attack to the SDN corresponds to an event or a recovery task for RAE, and there may be multiple refinement methods to recover from it. RAE+UPOM suggests to the Security Manager of ACR-SDN the best way to proceed. Our experiments show that integrating RAE+UPOM within ACRS improves the estimated time for recovery, efficiency, and retry-ratio with 95% confidence on a simulated SDN.

ACR-SDN configured RAE+UPOM to optimize a linear combination of efficiency and probability of success, and observed that the SDN can always recover after a finite number of attempts. This is because it is possible to recover from attacks by rebooting the network, if nothing else works. In future, it will be interesting to model scenarios with dead-ends that cannot be recovered from.

**Future work.** To leverage RAE+UPOM to achieve autonomous cyber responses, human domain experts are needed to develop the refinement methods in the hierarchical operational model. This requires them to understand RAE+UPOM, in order to effectively express their knowledge of the cyber domain and tune the system to work well in practice. Areas of future research are to ease this burden, to develop a framework or methodology that domain experts can use to encode their domain knowledge for the planning context, and to expand the research results from ACR-SDN to other IT systems.

Actions have cost estimates assigned to them in the operational model. Furthermore, a refinement method can also have a cost assigned to it, in which case this cost is added to the sum of the costs of the actions to arrive at an overall cost that is used for planning. However, sometimes a particular action may conflict with operational requirements, for example when blocking an attack must also block critical services. Capturing such conflicts and taking them into account when planning is another area of future research.

## Acknowledgments

This work has been supported in part by NRL grant N00173191G001 and ONR grant N000142012257. The information in this paper does not necessarily reflect the po-

sition or policy of the funders, and no official endorsement should be inferred.

## References

Berman, D. S.; Buczak, A. L.; Chavis, J. S.; and Corbett, C. L. 2019. A Survey of Deep Learning Methods for Cyber Security. *Information* 10(4): 122.

Bhuyan, M. H.; Bhattacharyya, D. K.; and Kalita, J. K. 2014. Network Anomaly Detection: Methods, Systems and Tools. *IEEE Communications Surveys Tutorials* 16(1): 303–336.

Despouys, O.; and Ingrand, F. 1999. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *5th European Conference on Planning (ECP)*, 280–292.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Gironza-Ceron, M. A.; Villota-Jacome, W. F.; Ordonez, A.; Estrada-Solano, F.; and Caicedo Rendon, O. M. 2017. SDN management based on Hierarchical Task Network and Network Functions Virtualization. In *2017 IEEE Symposium on Computers and Communications (ISCC)*, 1360–1365.

Han, Y.; Rubinstein, B. I. P.; Abraham, T.; Alpcan, T.; de Vel, O. Y.; Erfani, S. M.; Hubczenko, D.; Leckie, C.; and Montague, P. 2018. Reinforcement Learning for Autonomous Defence in Software-Defined Networking. *CoRR* abs/1808.05770. URL http://arxiv.org/abs/1808.05770.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers.

Ingrand, F. F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, 43–49.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *17th European Conference on Machine Learning (ECML)*, 282–293.

KP, S.; Alazab, M.; et al. 2020. A Comprehensive Tutorial and Survey of Applications of Deep Learning for Cyber Security. *TechRxiv* .

Lawal, B. H.; and Nuray, A. T. 2018. Real-time detection and mitigation of distributed denial of service (DDoS) attacks in software defined networking (SDN). In *2018 26th Signal Processing and Communications Applications Conference (SIU)*, 1–4.

Lee, S.; Kim, J.; Woo, S.; Yoon, C.; Scott-Hayward, S.; Yegneswaran, V.; Porras, P.; and Shin, S. 2020. A comprehensive security assessment framework for software-defined networks. *Computers & Security* 91: 101720.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL–the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mendes, L. D. P.; Aloi, J.; and Pimenta, T. C. 2019. Analysis of IoT Botnet Architectures and Recent Defense Proposals. In *2019 31st International Conference on Microelectronics (ICM)*, 186–189.

Mu, T.-Y.; Al-Fuqaha, A.; Shuaib, K.; Sallabi, F. M.; and Qadir, J. 2018. SDN Flow Entry Management Using Reinforcement Learning. *ACM Transactions on Autonomous and Adaptive Systems* 13(2). ISSN 1556-4665. doi:10.1145/3281032. URL https://doi.org/10.1145/3281032.

Ochoa-Aday, L.; Cervelló-Pastor, C.; and Fernández-Fernández, A. 2019. Self-healing and SDN: bridging the gap. *Digital Communications and Networks* ISSN 2352-8648. doi:https://doi.org/10.1016/j.dcan.2019.08.008. URL http://www.sciencedirect.com/science/article/pii/S2352864818302827.

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019. Acting and planning using operational models. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 7691–7698.

Patra, S.; Mason, J.; Ghallab, M.; Nau, D.; and Traverso, P. 2020a. Deliberative Acting, Online Planning and Learning with Hierarchical Operational Models. *arXiv preprint arXiv:2010.01909* .

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020b. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*. Best student paper honorable mention award.

Shrobe, H.; Laddaga, R.; Balzer, B.; Goldman, N.; Wile, D.; Tallis, M.; Hollebeek, T.; and Egyed, A. 2007. AWDRAT: a cognitive middleware system for information survivability. *AI Magazine* 28(3): 73–73.

Thorat, P.; Raza, S. M.; Nguyen, D. T.; Im, G.; Choo, H.; and Kim, D. S. 2015. Optimized Self-Healing Framework for Software Defined Networks. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, IMCOM '15. New York, NY, USA: Association for Computing Machinery. ISBN 9781450333771. doi:10.1145/2701126.2701235. URL https://doi.org/10.1145/2701126.2701235.

Yamaguchi, S. 2020. Botnet Defense System: Concept and Basic Strategy. In *2020 IEEE International Conference on Consumer Electronics (ICCE)*, 1–5.

Yoon, C.; Lee, S.; Kang, H.; Park, T.; Shin, S.; Yegneswaran, V.; Porras, P.; and Gu, G. 2017. Flow Wars: Systemizing the Attack Surface and Defenses in Software-Defined Networks. *IEEE/ACM Transactions on Networking* 25(6): 3514–3530.

Zargar, S. T.; Joshi, J.; and Tipper, D. 2013. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys Tutorials* 15(4): 2046–2069.

Zhang, L.; Yu, S.; Wu, D.; and Watters, P. 2011. A Survey on Latest Botnet Attack and Defense. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 53–60.