Naval Research Laboratory
Washington, DC 20375-5320

U.S. NAVAL RESEARCH LABORATORY

NRL/5540/MR—2022/2

ACRS4SDN: An Autonomous Cyber Response System for Software-Defined Networks

ALEXANDER VELAZQUEZ

BRUCE MONTROSE

MARGERY LI

JIM LUO

MYONG H. KANG

Center for High Assurance Computer Branch
Information Technology Division

SUNANDITA PATRA

DANA S. NAU

Department of Computer Science
College Park, MD

April 18, 2022

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
18-04-2022	NRL Memorandum Report	

4. TITLE AND SUBTITLE

ACRS4SDN: An Autonomous Cyber Response System for Software-Defined Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Alexander Velazquez, Sunandita Patra*, Bruce Montrose, Margery Li, Jim Luo,
Dana S. Nau*, and Myong H. Kang

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
6C14

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/5540/MR--2022/2

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
One Liberty Center
875 N. Randolph Street, Suite 1425
Arlington, VA 22203-1995

10. SPONSOR / MONITOR'S ACRONYM(S)

ONR

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

*Department of Computer Science, University of Maryland, 8125, Paint Branch Drive, College Park, MD 20742

14. ABSTRACT

Software-defined networks (SDNs) are susceptible to a wide variety of known and unknown cyberattacks. With adversaries that are capable of generating 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 cybersecurity experts to keep up with the necessary recovery and defense tasks. In this paper, we introduce ACRS4SDN, a system to monitor for, and quickly respond to attacks and failures that may occur in a SDN. An integral part of ACRS4SDN is its ability to autonomously recover using automated acting and planning, and it does so using a technique called hierarchical refinement. ACRS4SDN recovers a target system from faults and attacks by online planning using attack recovery procedures written as a hierarchical operational model. The autonomous responses orchestrated by ACRS4SDN considerably narrow the gap between cyberattacks and cyber defense, in terms of speed and volume, and we validate this through experimental results on a real SDN across a series of cyberattack scenarios.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Alexander Velazquez
U	U	U	U	36	**19b. TELEPHONE NUMBER** *(include area code)* (202) 404-4879

This page intentionally left blank.

CONTENTS

This page intentionally left blank

EXECUTIVE SUMMARY

In today's world, cyberattacks are dynamic, fast-paced, and high-volume, while most cyber responses are human speed. Therefore, keeping pace with existing and emerging cybersecurity threats is a challenging task. It is important to construct resilient computer systems that can autonomously protect against and recover from cyberattacks and system failures. In this paper, we chose software-defined networks (SDNs) as a sample class of systems to make resilient, because SDNs are relatively simple but an important class of systems to defend.

SDNs are susceptible to a wide variety of known and unknown cyberattacks. With adversaries that are capable of generating 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 cybersecurity experts to keep up with the necessary recovery and defense tasks. In this paper, we introduce ACRS4SDN, a system to monitor for, and quickly respond to attacks and failures that may occur in a SDN. An integral part of ACRS4SDN is its ability to autonomously recover using automated acting and planning, and it does so using a technique called hierarchical refinement. ACRS4SDN recovers a target system from faults and attacks by online planning using attack recovery procedures written as a hierarchical operational model. The autonomous responses orchestrated by ACRS4SDN considerably narrow the gap between cyberattacks and cyber defense, in terms of speed and volume, and we validate this through experimental results on a real SDN across a series of cyberattack scenarios.

This page intentionally left blank

ACRS4SDN: AN AUTONOMOUS CYBER RESPONSE SYSTEM FOR SOFTWARE-DEFINED NETWORKS

1. INTRODUCTION

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 [1–4], most system adaptation and recovery processes are ad-hoc, manual, and slow. Therefore, keeping pace with existing and emerging cybersecurity threats is a challenging task. It is important to construct resilient computer systems that can autonomously protect against and recover from cyberattacks and system failures. Software-defined networks (SDNs) provide advantages over traditional networks in terms of performance, agility, and monitoring. However, they also introduce a new attack surface that needs to be defended [5]. In this paper, we introduce ACRS4SDN, an autonomous cyber response system (ACRS) for SDNs.

The network is a key component in an enterprise IT system. 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. In general, network management architectures have two planes:

1. the data plane, where the traffic flows and network packets are forwarded; and
2. the control plane, which manages the packet routing process.

In traditional network architectures, these two planes are highly coupled and the control is decentralized, which can lead to complexity and lack of agility. SDN architectures address these issues by decoupling the two planes and having a centralized control plane, implemented using a set of controllers. However, this change in design comes with its own drawbacks when it comes to security. ACRS4SDN is implemented as the management plane in a SDN, which interacts with SDN components in the control plane and data plane to address security challenges through autonomous defense and recovery capabilities. Our focus in this paper is not on automated attack detection; rather, we focus on enabling autonomous responses once anomalies are detected using existing detection techniques. In our design and implementation of ACRS4SDN, we address the following challenges:

- Capturing expert cyber domain knowledge in a machine-understandable representation.
- Enabling real-time decision-making and recovery plan execution in response to attacks on a SDN.
- Providing situational awareness to human operators through a graphical user interface and human-understandable explanations of actions (*what* was done, and *why*).

Manuscript approved April 15, 2022.

The autonomous cyber response problem can be viewed as a real-time decision-making problem because an ACRS has to decide which actions (among several possibilities) have to be carried out in response to attacks in real time (see Section 4 for an example). AI planning and model-free or model-based reinforcement learning (RL) are techniques that are commonly used to solve decision-making problems.

To choose the appropriate technique to recover from a compromised state in a SDN, we analyze the following characteristics of the cybersecurity domain.

- A SDN system contains various components (switches, controllers) in different combinations and configurations. In typical RL techniques, the model needs to be retrained for every new configuration, which can add considerable overhead.
- The number of components in a SDN may be dynamic and change frequently. For example, the number of switches may change during operation due to malfunction, cyberattack, or moving target defense.
- There are no established rules between attackers and defenders regarding their actions. This discouraged us from using game-theoretic approaches.

Therefore, naïve application of RL to the cyber domain may not be effective due to the dynamic nature of this environment. In this paper, we will show how the above issues are addressed by ACRS4SDN in order to protect a SDN by using RAE (Refinement Acting Engine) + UPOM (UCT Planner for Operational Models). RAE+UPOM is an AI system especially designed to operate in a dynamic environment [6].

Unlike classical AI planning procedures (which are designed to run offline and search over a large space of actions, many of which are irrelevant to the problem at hand), RAE+UPOM supports fast online planning integrated with concurrent execution of recovery steps. It uses a hierarchical organization of problem-solving in which the higher levels represent general recovery procedures that are applicable to a wide variety of SDN-recovery problems, and the lower levels represent details that fit those general procedures to the specific details of the SDN-recovery problem at hand. This makes the problem tractable by focusing the search on the relevant parts of the search space.

RAE+UPOM consists of two subsystems: RAE [7], an *actor* that executes refinement methods (in our case, attack recovery procedures written by human experts); and UPOM, a *planner* that predicts how well each method will perform in the current situation. Given a task to perform, RAE will call UPOM to get advice on which refinement method to use. UPOM generates this advice quickly using Monte Carlo rollout techniques. When RAE evaluates the recommended method, it may include another task to perform. In such cases, RAE will call UPOM again to get advice on what method to use for that task.

This document is organized as follows. We start by discussing related work in Section 2. In Section 3, we describe the architecture of ACRS4SDN. In Section 4, we give an example SDN attack-and-recovery scenario. Then we give some background on refinement acting and planning (RAE+UPOM) and describe how it can help SDNs recover from cyberattacks in Section 5. In Section 6, we describe how expert knowledge of the cybersecurity domain is encoded into a hierarchical operational model suitable for planning. In Section 7, we explain how humans are kept informed via a graphical user interface and explanations of actions. We present experimental results in Section 8. Finally, in Section 9 we conclude and highlight some future research directions.

2. RELATED WORK

As AI and machine learning (ML) techniques advance and show success in many domains such as gaming and autonomous driving, some effort has been invested to apply AI and ML techniques to SDN management and cybersecurity. However, the field of applying AI and ML to autonomous cyber response is still in its infancy. AI planning has been applied to manage SDNs [8], self-healing of SDNs has been studied [9, 10], and RL has been applied to SDN flow rule management [11]. Applying deep learning to security mainly focuses on the detection of intrusions and malware [12, 13]. 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 SDN [14]. However, [14] did not consider typical cyberattacks to SDNs (see [5] 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 [15]) and network traffic anomalies [16]. However, little research has been done on autonomous cyber responses. In this paper, we do not focus on 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. 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 is based on an earlier system called PRS [17], which evaluated 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 [18], a planner that used state-space search techniques. UPOM’s Monte Carlo rollouts give it the ability to 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 [19], an algorithm for decision-making on Markov decision processes 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 Appendix (Section A). [20] discusses the adaptations we made to RAE+UPOM in order to use them in ACRS4SDN.

3. ARCHITECTURE

The architecture for ACRS4SDN consists of a number of components, organized in three layers that interact among themselves (see Figure 1). The *Presentation Layer* provides a graphical user interface (GUI) to keep humans in the loop with respect to the operation of the system (see Section 7 for detailed description). The *Security Layer* contains components that gather situational awareness and provide security services (e.g., monitoring and intrusion detection) for the rest of the system. The *Intelligent Planning Layer* deals with decision-making and is responsible for planning recovery procedures to be executed on the SDN.

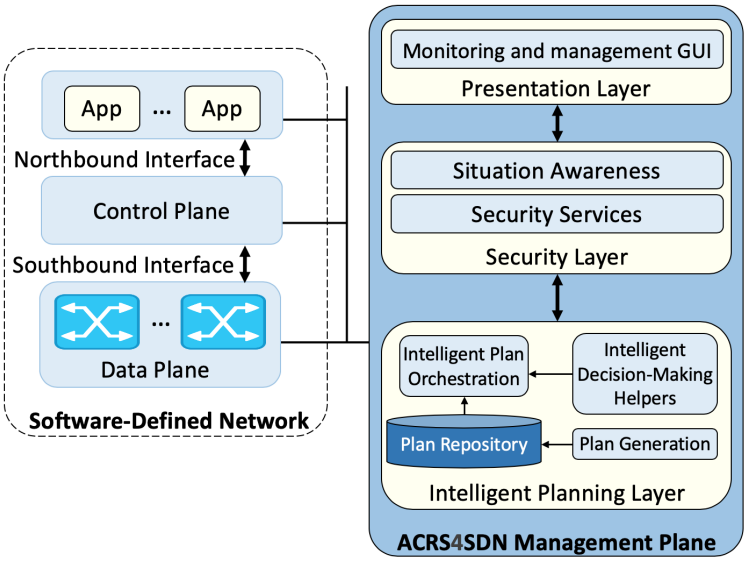


Fig. 1—Architecture of ACRS4SDN

All of these components reside in the SDN management plane and interact with the SDN components (switches, controllers), as shown in Figure 1.

Figure 2 illustrates the components and data flows necessary for the ACRS4SDN architecture.

- The *Management subsystem* contains the main components of ACRS4SDN that perform intelligent planning to remedy problems, orchestrate actions, and interface to human operators.
- The *Infrastructure subsystem* is the underlying platform responsible for provisioning the virtual machines (VMs) that host SDN controllers and switches. It also contains sensors that monitor the state of VMs.
- The *Control subsystem* comprises SDN components (e.g., controllers, switches) and sensors that monitor the components' status. Additionally, the control subsystem is responsible for executing actions over the controllers and switches.

The input to ACRS4SDN is state information, which is gathered from the SDN via sensors in the infrastructure and control subsystems and collected in the management subsystem. The output of ACRS4SDN are actions, which are orchestrated in the management subsystem and executed in the infrastructure and control subsystems. We describe the state space and action space in more detail in Section 6.

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 ACRS4SDN components via publish-subscribe messaging (Pub-Sub) and remote procedure calls (RPC).

Our architecture is designed to be flexible and extendable, in order to easily support future changes in requirements. For example, if a new cyberattack is discovered, a detection procedure can be added to the security layer and one or more corresponding mitigation procedures can be added in the planning layer. If this requires additional state to be monitored, agents in the control plane already exist and can be extended to report the new state variables.

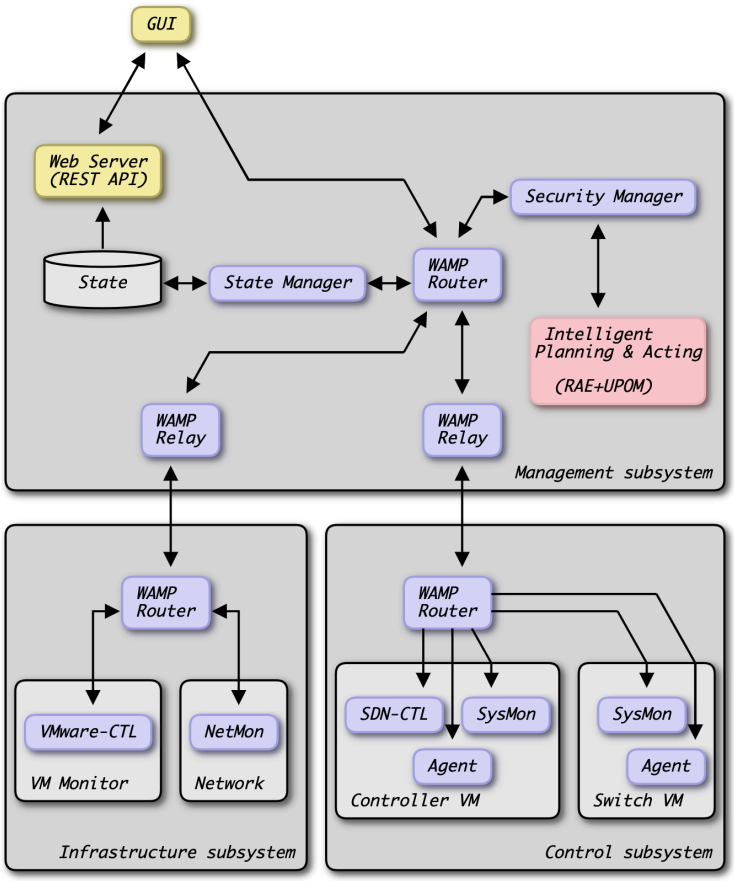


Fig. 2—Detailed architecture and data flows between components in ACRS4SDN

3.1 Components of ACRS4SDN

WAMP Router. A central WAMP router in the management subsystem allows the State Manager, Security Manager, and WAMP Relays to communicate.

State Manager. The State Manager keeps track of the 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.

Security Manager. The Security Manager receives system statistics, log messages, and alerts from the control and infrastructure subsystems. It monitors the health of each component and the overall health of the network, and uses configurable thresholds to determine when a task needs planning.

Intelligent Planning and Acting Module. RAE+UPOM is called upon by the Security Manager when it has a recovery task, for which planning is required. 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. RAE gets all of its information about the system's state directly from the Security Manager and relies on the Security Manager to carry out the planned actions on the system and provide feedback. Whenever RAE has planned an action for the Security Manager to execute, it puts it in a command execution queue. The Security

Manager continuously monitors the command execution queue for incoming commands. Once it receives a command, it executes it, and sends information back to RAE about whether the command succeeded or failed, along with the new state. We describe the communication in more detail in Section 3.2.

Web Server. The Web Server provides a RESTful interface to the current state of the SDN. When requested by the GUI, it queries the state information in the database and returns a snapshot of the system state.

GUI. The GUI provides human operators with a real-time view of the state of the system. It requests data from the Web Server and also receives real-time updates from other components via the WAMP Router.

WAMP Relay. The WAMP Relays allow for controlled access to components in the SDN (control subsystem) and virtual machine platform (infrastructure subsystem), 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 subsystems: SDN-CTL, SysMon, NetMon, Agent, etc.

3.2 Communication between Security Manager and RAE

Most of the components communicate using the aforementioned WAMP protocols (Pub-Sub and RPC). However, the Security Manager and RAE are more tightly coupled, so their communication mechanism warrants some additional explanation. Communication between the Security Manager and RAE occurs using three shared queues:

- **Task queue**: After the Security Manager detects an attack, it puts an attack event or a recovery task, and the state, on the task queue. The task stays in the queue until RAE reads from it.
- **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 the command succeeded or failed and the next state of the SDN in the command status queue. RAE reads this information and updates the state accordingly.

The command execution queue and the command status queue work together in a back-and-forth manner. RAE plans for a recovery task using UPOM. UPOM returns the first command in the plan. RAE puts this command in the command execution queue. The Security Manager continuously monitors the command execution queue for incoming commands. Once it sees a command, it pops the command from the queue, and executes the command on the target system, the SDN. Once the command is done executing, the Security Manager puts the updated state (the details of what information the state contains is described in Section 6), and information about whether the command has succeeded or failed, on the command status queue. RAE receives this information and again re-plans (see Section 5). It then sends the next command to execute to the Security Manager.

4. EXAMPLE OF SDN RECOVERY SCENARIO

SDNs are vulnerable to not only traditional network attacks such as Denial of Service (DoS) attack but also various SDN-specific attacks. A number of SDN-specific attacks are discussed in [5, 21]. Typically,

avenues of attack 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 SDN components.

In addition to known attacks, SDNs may also be vulnerable to unknown (i.e., zero-day) attacks, as well as system faults. In these cases, we may not know the precise cause of the failure (our focus in this paper is not on attack detection); however, we can expect the symptoms to manifest as a change in system state, that can be discovered via probing actions. For example:

1. Switch malfunction: switch exhibits unexpected behavior due to an internal error or some attack from outside.
2. Controller malfunction: resource consumption of controller goes out of bounds.

In a PACKET_IN flooding attack, one or more malicious hosts continuously send traffic to an unknown (and possibly randomized) destination address. Every time a switch receives a packet that it does not know how to handle, it sends an OpenFlow PACKET_IN request to the controller. Since the controller does not know the location of the destination address, it instructs the switch to flood the packet to all output ports. Over time, if the volume of packets is large enough, the flood of PACKET_IN requests eats up control plane bandwidth and can cause denial of service of the controller. It can also consume resources on the switch that a malicious host is connected to, and the side-effects can be felt network-wide as long as the controller remains unresponsive.

After the attack has begun, 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: increased controller host table size, control plane network bandwidth saturation, increased control plane network latency, and increased CPU usage on the switch connected to the malicious host.

When it comes to monitoring, probing for additional clues, and ultimately responding to abnormal system states, our high-level approach in ACRS4SDN is the following: regularly gather system statistics in real time to make preliminary determination about whether any abnormal symptoms exist; when the system or a component is in an abnormal state, probe for additional information (some of which may not be suitable for real-time collection) to narrow down root causes and identify abnormal or malicious components; and finally plan and execute a sequence of actions to remedy the problem.

To mitigate this PACKET_IN flooding 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.

This scenario informally illustrates the methodology of an attacker and the thought that would go into recovering the SDN by a human expert. Later, in Section 6, we show how ACRS4SDN captures this type of

expert cyber domain knowledge in the form of a hierarchical operational model. In Section 8, the performance of ACRS4SDN will be shown to recover the SDN through refinement acting and planning.

5. REFINEMENT ACTING AND PLANNING

This section gives an overview of RAE, UPOM and the hierarchical operational model representation. For further information about them, see [6]. Together, RAE and UPOM form an acting and planning system in dynamic environments with exogenous events. [6] shows that RAE +UPOM improves the performance in a diverse set of simulated environments. From the point of view of the acting engine, RAE, the attacks on a SDN are modeled as exogenous events happening in the SDN environment. The operational models tell the actor how to perform tasks. Its basic ingredients are *tasks*, *events*, *commands*, and *refinement methods*.

Consider the example in Section 4 of recovering from a PACKET_IN flooding attack. In order to help the SDN recover from PACKET_IN flooding of a controller, RAE takes as input programming procedures written by human experts. These procedures are called refinement methods and describe different ways of recovering from the attack. A *task* is an activity for the actor to perform. An *event* is an exogenous occurrence in the environment that the actor needs to recover from. A task or event can have one or more refinement methods. A *refinement method* is of 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. Here are three refinement methods to recover from PACKET_IN flooding described in Section 4.

The first method, m1_ctrl_clearstate_besteffort(id), simply clears the host table in the controller, which may be enough if the attacker has stopped sending new packets and the only lingering cause of resource exhaustion is an inflated host table in the controller. A refinement tree for packetIn-flooding(id) using m1_ctrl_clearstate_besteffort(id) is shown in Figure 3.

```
m1_ctrl_clearstate_besteffort(id)
  event: packetIn-flooding(id)
    pre: None
  body: if not is_component_type(id, 'CTRL'):
            fail
        else: clear_ctrl_state_besteffort(id)
```

The second method, m2_ctrl_clearstate_fallback(id), also clears the host table in the controller, but does so in a higher assurance way, which may take longer but is less likely to fail. A refinement tree using m2_ctrl_clearstate_fallback(id) is shown in Figure 4.

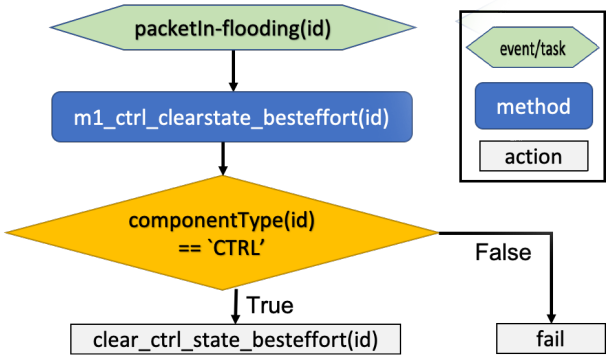


Fig. 3—A refinement tree for the event packetIn-flooding(id) using the refinement method m1_ctrl_clearstate_besteffort(id).

m2_ctrl_clearstate_fallback(id)
event: packetIn-flooding(id)
body: if not is_component_type(id, 'CTRL'):
fail
else: clear_ctrl_state_fallback(id)

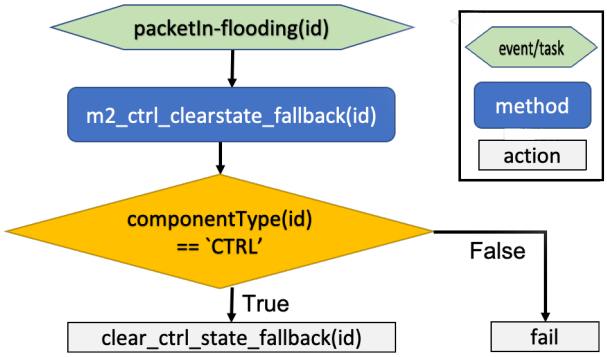


Fig. 4—A refinement tree for the event packetIn-flooding(id) using the refinement method m2_ctrl_clearstate_fallback(id).

The third method, m3_ctrl_mitigate_pktinflood(id), searches for switches which have been marked as unhealthy by the ACRS4SDN'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 in Figure 5.

```
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 new switch
                    add_switch (s_id)
                    # Move critical hosts from
                    unhealthy switches
                    move_critical_hosts(s_id, s_id + '-new')
                # Fix unhealthy switch
                fix_switch(s_id)
        # Clear controller state
        clear_ctrl_state_besteffort(id)
        # Check whether controller is now healthy
        if not is_component_healthy(id): fail
```

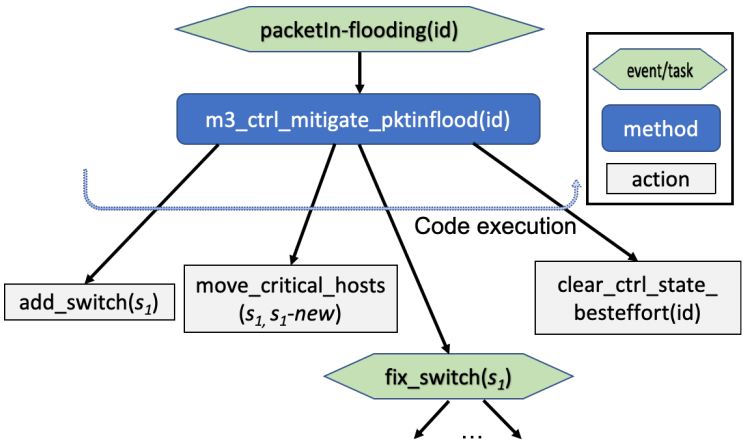


Fig. 5—A partial refinement tree for the event packetIn-flooding(id) using the refinement method m3_ctrl_mitigate_pktinflood(id). fix-switch(s_1) is a sub-task that should further be refined.

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

- S is the set of states the SDN may be in, e.g., componentType(id), componentHealth(id) $\in S$;
- $\mathcal{T}$ is the set of events (attacks) and recovery tasks that ACRS4SDN may have to deal with, e.g., packetin-flooding(id), fix-component(id) $\in \mathcal{T}$;
- $\mathcal{M}$ is the set of methods for handling tasks or events in $\mathcal{T}$, e.g., m1_ctrl_clearstate_besteffort(id), m2_ctrl_clearstate_fallback(id), m3_ctrl_mitigate_pktinflood(id) $\in \mathcal{M}$;

- $\mathcal{A}$ is the set of primitive actions or commands that can be executed on the SDN, e.g., clear_ctrl_state_fallback(id), add_switch(s) $\in \mathcal{A}$.

Acting and planning. The deliberative acting problem for ACRS4SDN can be stated informally as follows: given Σ 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) τ 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 5. ACRS4SDN 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 τ_0 to RAE is represented via a *refinement stack* σ, which keeps track of how much further RAE has progressed in recovering from τ_0. τ_0 can decompose into several sub-tasks following a refinement method. The refinement stack is a LIFO list of tuples $\sigma = \langle(\tau, m, i), \ldots, (\tau_0, m_0, i_0)\rangle$, where τ is the deepest current sub-task in the refinement of τ_0, m is the method used to recover from τ, i is the current instruction in $body(m)$, σ is handled with the usual stack push, pop and top functions.

When RAE addresses a task or event τ, it must choose a method m to handle τ. 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 σ 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.

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, RAE uses the UPOM planner.

AI planning systems typically represent actions using *descriptive models* written in a language such as PDDL [22, 23]. 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. 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. 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 τ and a state s approximately optimal for a utility function U. It is a UCT-like [19] Monte Carlo tree search (MCTS) procedure over the space of refinement trees for τ. The mapping from UPOM's search to a UCT search is rather complicated; for details see [6].

Utility function for UPOM. UPOM can optimize different utility functions, such as the acting efficiency (reciprocal of the estimated time) or the probability of success. In this paper, we focus on optimizing the *cost-effectiveness* of methods, which is a linear combination of efficiency and probability of success. For the cyber defense domain, we define cost-effectiveness to be a utility function with the following properties:

1. The value is 0 if the action ultimately leads to failure (SDN can't recover).
2. The value is inversely proportional to the cost; that is, a more expensive action corresponds to a lower value.

If we are interested in defending and recovering a SDN from cyberattacks, we need to select actions that maintain or return the system to a healthy state, and do so with minimal resources spent. If the task at hand is to repair a given component (switch or controller), we are not interested in actions that get us close but ultimately fail at completing the task, no matter how much cheaper those actions are compared to the alternatives. We would rather select a more expensive course of actions that succeeds at the recovery task. Having said that, we do want to minimize cost. For example, if a switch's flow table is corrupted, rebooting the switch can be effective but would cause considerable downtime, whereas flushing and repopulating the flow table could be a cheaper (faster) way to achieve the result.

Our motivation for defining our utility function, cost-effectiveness, in this way is to guide UPOM in its Monte Carlo tree search towards effective-but-cheap solutions, while ignoring ineffective solutions and de-prioritizing expensive ones.

Cost Effectiveness. Let a method m for a task τ have two sub-tasks, τ_1 and τ_2, with costs c_1 and c_2 respectively. The cost-effectiveness of τ_1 is $u_1 = 1/c_1 + \alpha$ if τ_1 succeeds, and 0 if τ_1 fails. α is a scaling factor to weight the probability of success with the cost. The cost-effectiveness of τ_2 is $u_2 = 1/c_2 + \alpha$ if τ_2 succeeds, and 0 if τ_2 fails. The cost-effectiveness of accomplishing both tasks is

$$\begin{array}{c} 1/(c_1 + c_2) + \alpha, \text{ if both } \tau_1 \text{ and } \tau_2 \text{ succeed,} \\ 0, \text{ otherwise.} \end{array} \tag{1}$$

Thus, the incremental cost-effectiveness composition is:

$$\begin{array}{l} u_1 \oplus u_2 = u_2 \text{ if } u_1 = \infty, \text{ else} \\ \qquad\quad u_1 \text{ if } u_2 = \infty, \text{ else} \\ \qquad\quad 0 \text{ if } u_1 = 0 \text{ or } u_2 = 0, \text{ else} \\ \qquad\quad \dfrac{(u_1 - \alpha)(u_2 - \alpha)}{u_1 + u_2 - 2\alpha} + 1. \end{array} \tag{2}$$

$u_1 \oplus u_2 = 0$, meaning that τ fails with method m. Note that Formula 2 is associative. When using cost-effectiveness as a utility function, we denote $U(\text{Success}) = \infty$ and $U(\text{Failure}) = 0$. In our experiments, we used $\alpha = 0.05$.

6. DESIGNING OPERATIONAL MODELS FOR RAE+UPOM

RAE+UPOM operates on hierarchical operational models of tasks and refinement methods. A task is an activity that needs to be accomplished. A refinement method describes one way of accomplishing a task, and it can decompose into a combination of sub-tasks and actions. RAE+UPOM has been applied to a variety

of domains; in each case, a domain expert needs to encode their domain knowledge using this language of operational models.

In order to leverage RAE+UPOM in ACRS4SDN, we need to define the state space, action model, and refinement methods that RAE+UPOM uses. We describe each of them as follows.

State definition. State information is continually collected by the Security Manager from sensors in the infrastructure and control subsystems (Figure 2). When an attack is detected that requires a response to be planned and executed, the state is shared with RAE. Refinement methods in the operational model can inspect this state (e.g., to find out which methods are applicable). The actions are sent by RAE to the Security Manager. The Security Manager executes the actions on the SDN and sends the results back to RAE.

The state consists of two top-level Python dictionaries: *components* and *hosts*. In order to secure a SDN, ACRS4SDN deals with two types of components: controllers and switches. The *components* dictionary maps from component IDs properties of the component (*id*, *type*, *CPU/Mem*, etc.).

The *hosts* dictionary contains information about the data plane hosts that are served by the SDN switches. Not all hosts are necessarily captured in the state at all times, but sometimes it is important to have some information about some subset of hosts, for example those currently serving a mission-critical purpose.

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 due to a variety of reasons: an unexpected failure or outage, executing a course of action that includes adding or removing a component, performing moving target defense, etc. Our state representation is able to handle this dynamism. 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 refinement methods of the operational model, which are designed with this state representation in mind and enable effective planning no matter what the current system configuration looks like.

Task hierarchy and refinement methods. When designing the operational model, we desire a flexible model that does not need to be altered or updated when the requirements of the system change at runtime, nor when the environment changes due to adding or removing components. The operational model should be generic enough to handle the dynamic nature of the SDN. Since we know that the system can change over time (components are added or removed, IT equipment is upgraded, etc.), we want to avoid the need to change the operational model every time that happens.

To design the operational model for ACRS4SDN, we take a bottom-up approach. Every IT system is composed of various reusable components that can be viewed at different levels of abstraction: the low-level hardware, possibly virtual machines (VMs) running on that hardware, and software processes to carry out the business logic and serve the end-user. A SDN is composed of controllers and switches, each of which is a software process, possibly running in a VM, running on hardware. In order to defend the SDN and recover from failures, we need to perform actions on these components, such as:

- add flow rule to block traffic from a host,
- clear out controller host or switch table,

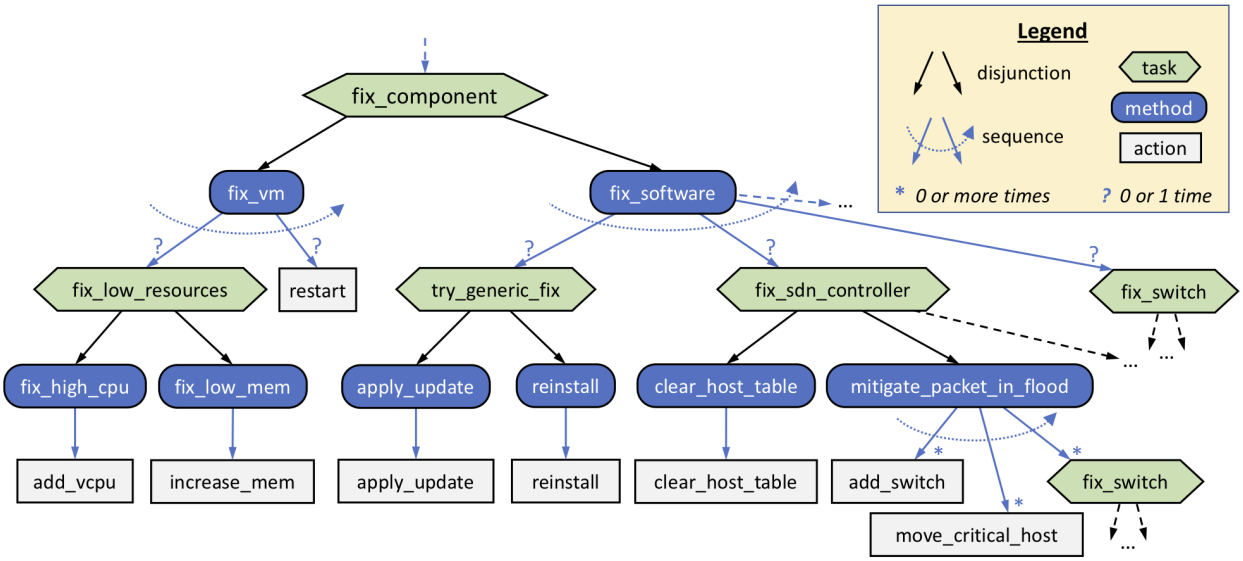


Fig. 6—This portion of the operational model is used to recover a SDN controller. The tasks serve as entry points into the planning process and map to one or more refinement methods. A refinement method is implemented as a Python function; thus, it can use the full expressive power of conditional statements, control structures, etc., to call sub-tasks and/or actions. The potentially complex logic of a refinement method is simplified in this representation, but some logic (optional or repetitive calls) are indicated using a symbol on the corresponding edge. Some portions of the operational model are omitted, which is indicated by an ellipsis or dashed edge.

- clear out switch flow table,
- move host from one switch to another, or,
- increase amount of memory available to the software process.

The refinement methods are where most of the domain knowledge is incorporated. The root cause of a symptom may not always be obvious from the current state information, and often we require some probing to get the necessary information to plan effectively and decide how to proceed. For example, in a SDN, the controller may become unresponsive or slow down enough to cause the system to stop handling new requests. We may detect that the CPU utilization of the controller and one or more of the switches is very high. If the controller is running in a VM, we could add virtual CPUs (VCPUs) or additional memory to alleviate the problem, but this solution may not be effective long-term.

Rather, we can probe to detect specific conditions in the state that could enable us to choose a more effective solution. The situation described above could be caused by the PACKET_IN flooding attack described in Section 4. Monitoring basic component statistics, such as CPU and memory utilization, is relatively inexpensive, so these state variables are updated in near real time at regular intervals. The Security Manager uses these statistics to know when to begin planning a response for a cyberattack or system failure. If we determine the target system to be in an abnormal or insecure state, then we require corrective action to return the system to a normal and healthy state. However, to gain better situational awareness, some probing must be done. This is because we cannot update every state variable in real time, or even at regular intervals, if the act of measuring or reading the data in question is relatively time-consuming or expensive. For example,

the list of active applications in the SDN controller may not be a necessary piece of information in every recovery scenario, so we leave that variable undefined in the state until a rudimentary root cause analysis determines that it may be needed for subsequent decision-making. Taking this phased approach to gathering state information allows ACRS4SDN to balance system performance with information completeness. We use near real-time system monitoring to establish a baseline state, which can be used by the Security Manager to decide when to initiate planning (Security Manager sends a task to RAE), and then probe to narrow down more precisely which state we are in.

In the PACKET_IN flooding attack, we first see CPU usage in the controller spike and remain high. One or more switches may report the controller as unresponsive. All of this can be seen in the baseline state. A task is passed from the Security Manager to RAE to fix the controller. The operational model contains various generic refinement methods to fix the controller, or more generally a software process, or an underlying VM, etc. Our domain knowledge tells us that the elevated resource usage in the controller could be caused by an attack. We probe for some additional state information:

- Controller host table size
- Control plane network bandwidth saturation
- Control plane network latency
- Abnormal network activity of any hosts

If one or all of these are problematic, then we may have some additional confidence that this is a PACKET_IN flooding attack. Often, a long-term fix for symptoms seen in one component may require performing actions on other components. This is one such example, where symptoms first appear in the controller but also require actions (probing and corrective) in one or more switches to fully address the problem. This type of expert knowledge is encoded by domain experts as refinement methods in the operational model. We iterate through the switches connected to the controller and see if any of them also have a high CPU load. If so, we can do further probing to see if any of them are sending a large volume of OpenFlow PACKET_IN requests. Finally, once the root cause is identified, we can plan for an effective mitigation. For example, creating a new switch, moving mission-critical hosts to the new switch, and then planning for an additional sub-task of fixing the old switch.

The portion of the hierarchical operational model described here can be seen in Figure 6.

6.1 Action and environmental model

Each low-level action is modeled in the operational model and has a counterpart on the execution platform (the SDN via the Security Manager of ACRS4SDN), so that when an action is passed by RAE to the Security Manager, it can be sent to an agent in the infrastructure or control subsystem and executed (Figure 2). These actions have relative costs associated with them (defined by experts in the operational model between 1 and 60), which are estimates of how long it will take to implement the action on the target system. For example: restarting a VM 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. This cost in conjunction with UPOM's search mechanism will guide ACRS4SDN towards cost-effective actions.

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.

7. INTERFACE TO HUMAN OPERATORS

ACRS4SDN provides a GUI that an operator can use to declare critical operations. ACRS4SDN captures critical operations in terms of pairs of network addresses that must be able to communicate. These required operations are translated into network intents, which are then loaded into the SDN controller with the push of a button in the ACRS4SDN GUI. Assets (e.g., switches, ports) and packets between critical hosts will be treated as high-priority ones that need to be protected. Other non-critical assets may be restored later than high-priority assets. Also, traffic from non-critical hosts may be blocked temporarily to support critical communication. A screenshot of this interface is shown in Figure 7.

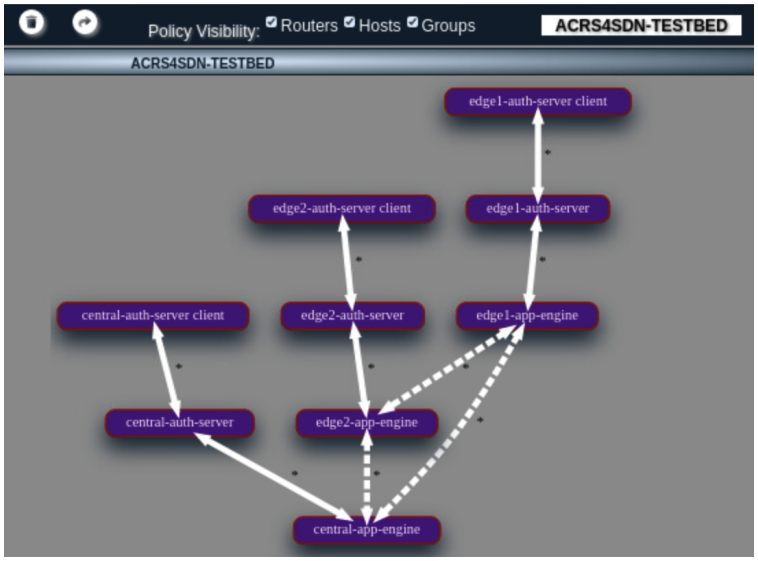


Fig. 7—A screenshot of the ACRS4SDN GUI, showing the interface used by human operators to declare critical operations.

ACRS4SDN acts autonomously when there is no human operator. However, it also provides a GUI to present its actions and rationale for the actions. For this purpose, ACRS4SDN provides three other runtime interfaces in the GUI:

1. a network panel to graphically show the components and overall health of the system,
2. a notification panel to display human-understandable explanations of the health status when anomalies or cyberattacks are detected, and

3. an action panel to display what actions have been performed to defend against cyberattacks or to recover the system from abnormal state.

A screenshot of the GUI is shown in Figure 8. The main network panel at the top shows control plane components (one SDN controller and two switches are visible) and each component's fill color indicates its current health as determined by its resource utilization (on a scale from green being healthy, to red being unhealthy). Below the switches are the data plane hosts (some are outlined in red to denote them as critical), with edges indicating what switch port each host is connected to.

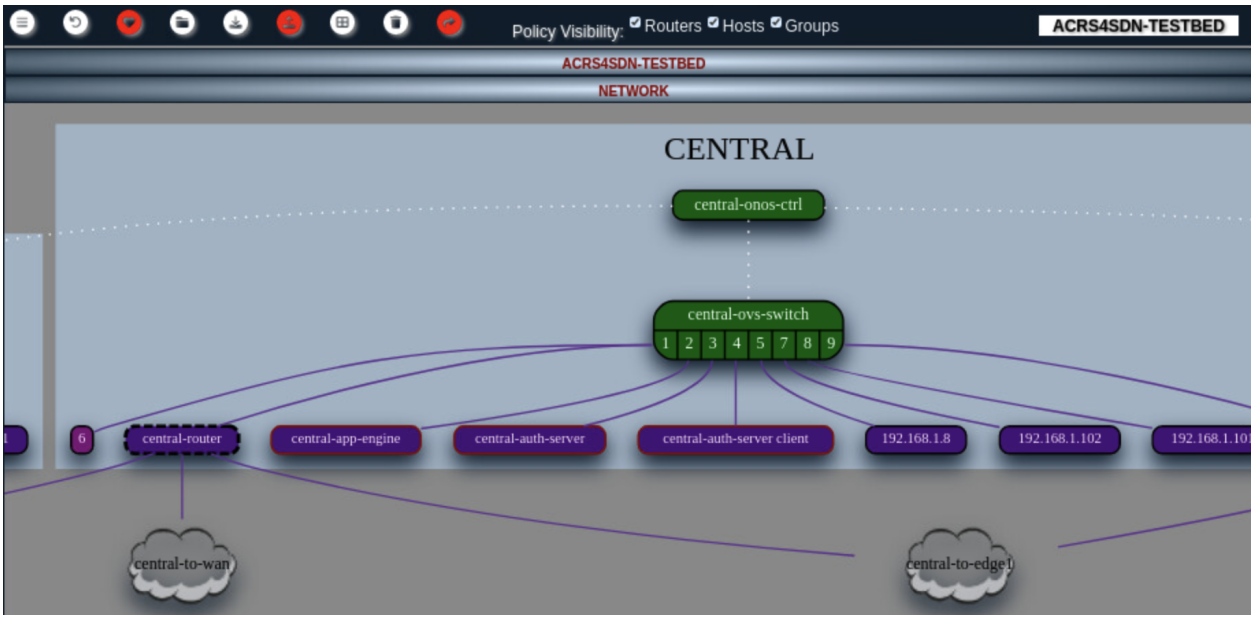


Fig. 8—A screenshot of the ACRS4SDN GUI, showing the current state of the SDN in the network panel. Panels for notifications and an actions log are also shown.

The Security Manager monitors the state of the SDN and submits tasks to RAE+UPOM when an attack is detected, the system is in an unhealthy state, etc. This may be a very high-level task, e.g., "fix SDN", that eventually boils down to one or more low-level actions after some probing has been done and planning has completed. However, the high-level context is an important piece of information when explaining the low-level actions that have been selected, since it gives clues as to *why* the action was necessary, and not just *what* is being done about it.

Therefore, our hierarchical operational model is designed to support the passing of context alongside tasks and refinement methods, and human-understandable explanations alongside actions. The following sequence of events occurs at runtime:

1. Security Manager passes a task with high-level context that triggered the planning process to RAE+UPOM.
2. When a refinement method is selected, it has access to the high-level context. If the refinement method calls another task, it appends the local context for this decision to the high-level context and passes it to the new task. If the refinement method calls an action, it appends a statement indicating what the action is intended to accomplish in this local context and passes it along with the action.

3. Security Manager receives an action, along with the generated explanation. This explanation is written to a log file and also sent as a notification to the GUI, where it is displayed for the human operator to read.

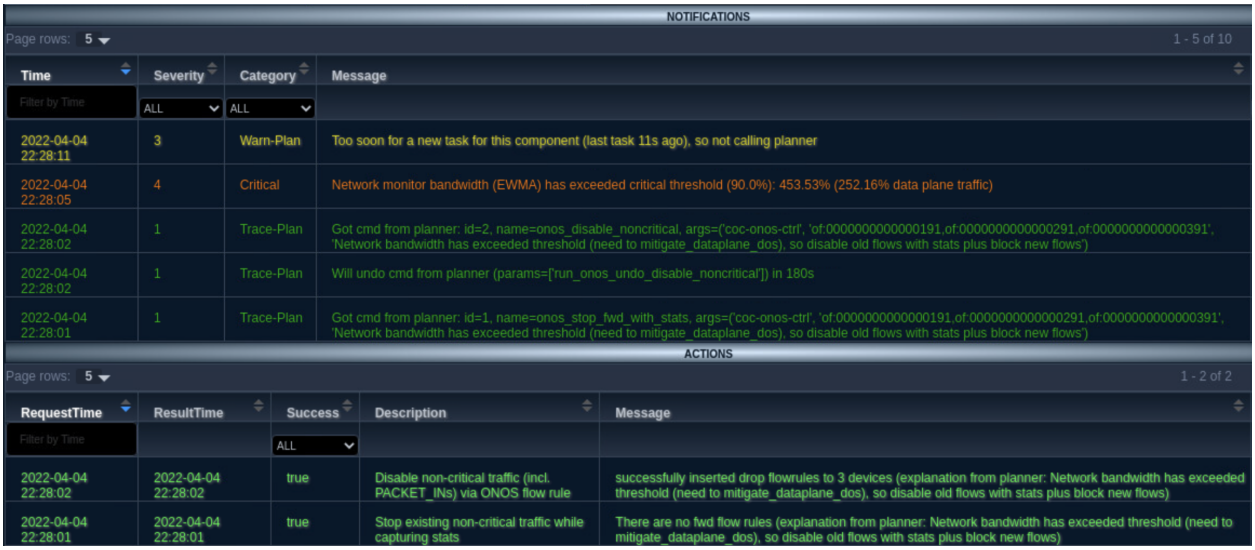


Fig. 9—A snapshot of the ACRS4SDN GUI during an attack and recovery scenario, where the notifications and actions panels are populated with human-understandable explanations and context.

An example of these explanations can be seen in the screenshot of the GUI in Figure 9. In this example, the high-level context that initially led the Security Manager to call the planner was that network bandwidth had exceeded a threshold, leading to the suspicion of a data plane DoS attack. As a result, two actions were ultimately taken: existing non-critical traffic (as defined by the human operator, see Figure 7) was stopped, and future non-critical traffic was temporarily disabled network-wide.

8. EXPERIMENTAL EVALUATION

In order to measure the performance of ACRS4SDN and validate that it achieves the required goals, we set up an experimental testbed environment and measured the performance of our recovery and responses across various cyberattack scenarios.

Our testbed models a SDN that spans three widely separated network nodes: a central node and two “edge" nodes. Each node contains an OpenFlow switch, and the central node also contains the SDN controller. There is a low-bandwidth, high-delay link between each edge node and the central node. These links allow 1 Mbps down and 0.5 Mbps up from the central node to the edge node with 400 ms round-trip delay, and they carry all of the data plane as well as control/management plane traffic.

We implemented our testbed on a VMware ESXi hypervisor with the following VMs:

- SDN controller

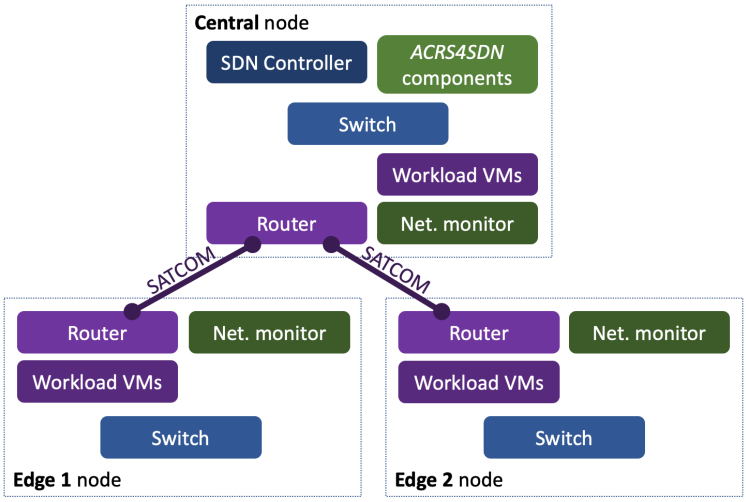


Fig. 10—An overview of the VMs deployed in three network nodes that constitute our testbed.

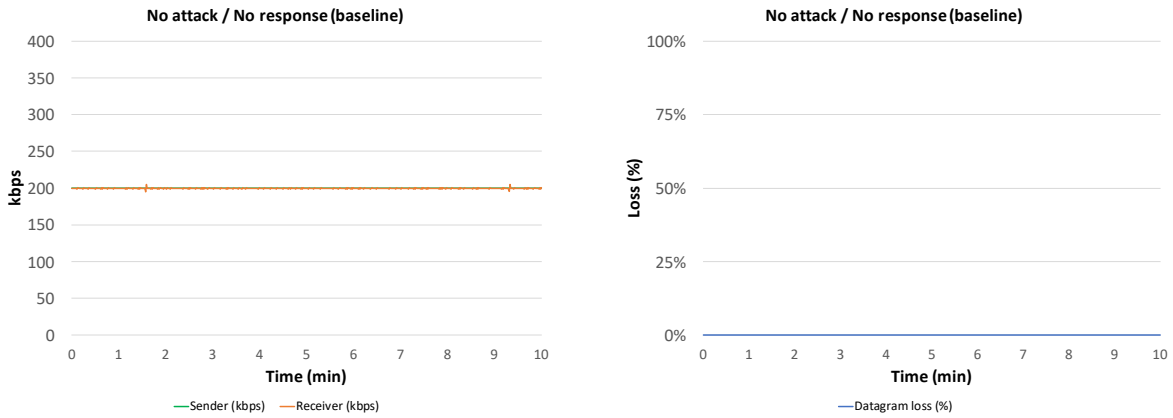
- Central switch
- Edge 1 switch
- Edge 2 switch
- ACRS4SDN components
- SATCOM emulator & router
- Central network monitor
- Edge 1 network monitor
- Edge 2 network monitor
- plus, a number of workload VMs for generating background traffic and attack traffic

Each VM utilized Debian 11.2 as its underlying OS. The SDN controller was running ONOS 2.7.0 and the switches were running Open vSwitch 2.10.7. The SATCOM links were emulated using the network emulation capabilities included with the Linux traffic control module (tc-netem).

To measure the impact of the attacks and our recovery, we transmitted a constant rate UDP flow at 40% of the bottleneck rate between a workload VM in an edge node (“sender") and a workload VM in the central node (“receiver"), using iperf3 (version 3.9). We established a baseline when the network was not under load, and then used that for comparison for the attack scenarios.

See Figure 11 for representative plots of this baseline. We see all traffic from the sender successfully arrive at the receiver. There are no lost datagrams. These performance characteristics held true across 10 trials: an average receiver throughput rate of 200.00 kbps, a total of 0 lost datagrams, and average jitter of 0.16 ms.

For our cyberattack scenarios, we launched three different types of cyberattacks:



(a) (b)

Fig. 11—(a) Sender/Receiver throughput rates (kbps) vs. time, and (b) datagram loss (%) vs. time, for baseline scenario (background constant-rate UDP flow with no attack and no response).

- Data plane denial of service (DoS) attack (high-rate UDP flow)
- PACKET_IN flooding attack[5]
- Flow Rule flooding attack[5]

8.1 Scenario 1: Data Plane DoS attack

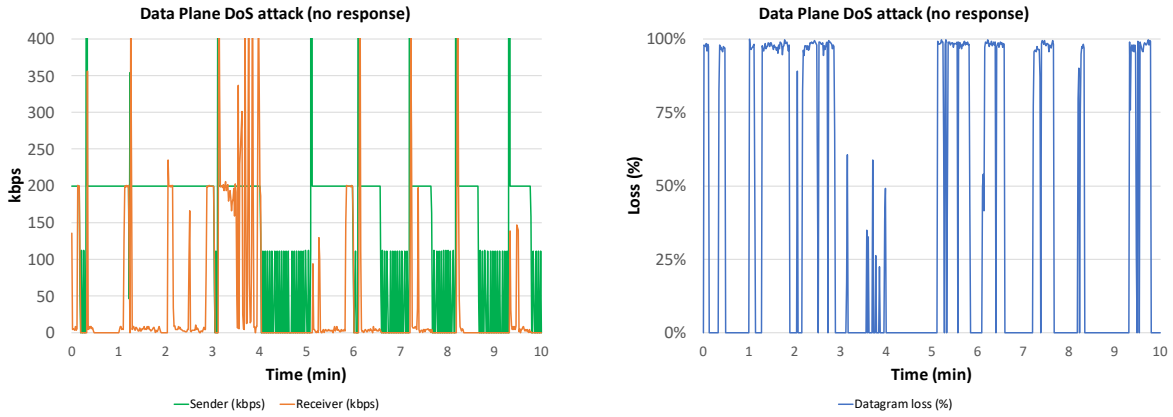
In the data plane DoS attack, a malicious host on the data plane transmits a stream of UDP packets at the maximum possible rate, directed towards an IP address in a remote node (thus forcing the traffic across the bottleneck link).

With ACRS4SDN disabled (i.e., no responses being performed), the data plane DoS attack causes considerable disruption to the network. See Figure 12 for sender and receiver throughput rates, as well as datagram loss percentage over time. Across 10 trials, the average receiver throughput rate was 44.45 kbps, 76.70% of datagrams from our test flow were lost, and average jitter was 6.88 ms.

On the other hand, when ACRS4SDN was enabled, it was able to recover the SDN from the data plane DoS attack quickly and effectively. There was some data loss initially before any action could be taken, but the response was decisive and blocked the attack completely until the temporary restrictions were automatically removed after 6 minutes.

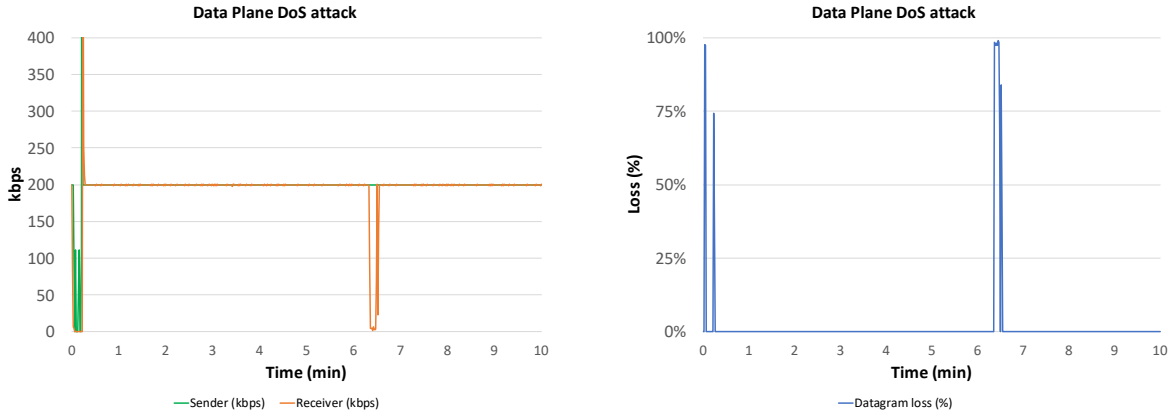
See Figure 13 for sender and receiver throughput rates, as well as datagram loss percentage over time. Over the entire experiment, averaged across 10 trials, the receiver throughput was 189.51 kbps, 5.25% of datagrams were lost, and jitter was 0.76 ms.

Without ACRS4SDN running, the data plane DoS attack caused nearly 15 times higher datagram loss. With ACRS4SDN enabled, the receiver throughput rate was more than 4 times higher.



(a) (b)

Fig. 12—(a) Sender/Receiver throughput rates (kbps) vs. time, and (b) datagram loss (%) vs. time, for data plane DoS attack without response.



(a) (b)

Fig. 13—(a) Sender/Receiver throughput rates (kbps) vs. time, and (b) datagram loss (%) vs. time, for data plane DoS attack with response.

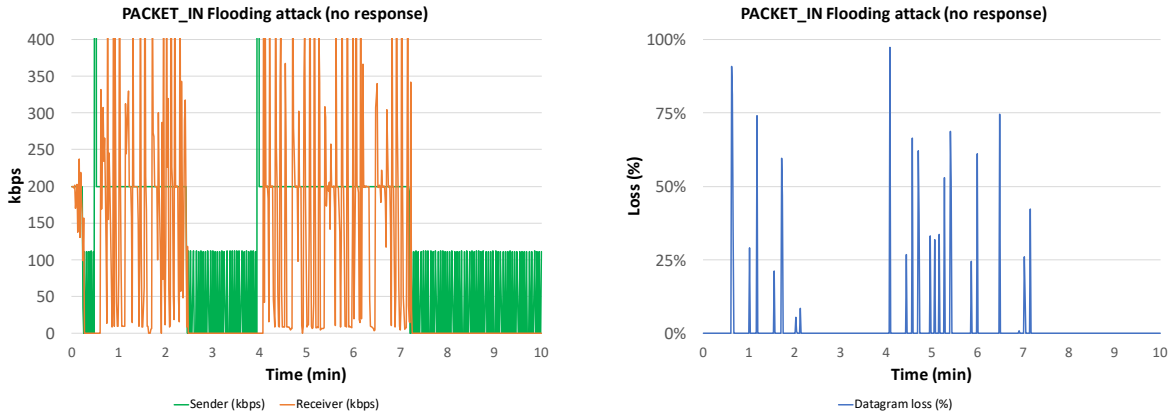
8.2 Scenario 2: PACKET_IN flooding attack

During a PACKET_IN flooding attack, a malicious host on the data plane transmits packets with randomized source and destination MAC addresses. When these arrive at the switch, they do not match any existing flow rules and must be sent to the SDN controller via an OpenFlow PACKET_IN message. Thus, this type of DoS attack is an amplification attack. The effects can typically be felt throughout the network as the SDN controller's resources are exhausted. Furthermore, if no action is taken, the network can continue to suffer as long as the controller's host table remains polluted with spurious entries. In our particular testbed setup, the low-bandwidth bottleneck links can also be overwhelmed with little effort on the part of the attacker.

Figure 14 shows throughput rates and datagram loss during a PACKET_IN attack when ACRS4SDN is disabled, while Figure 15 shows those results when ACRS4SDN is enabled.

In the face of a PACKET_IN flooding attack, ACRS4SDN was able to stop the attack and recover the SDN within a very short period. Typically, recovery involves not only blocking malicious PACKET_IN flows, but also clearing the SDN controller's polluted host table, so there can be some lingering effects in the network while the process completes.

Across 10 trials, the average receiver throughput rate was 108.65 kbps with ACRS4SDN disabled, compared to 167.07 kbps with ACRS4SDN enabled. The average datagram loss was 40.97% with ACRS4SDN disabled vs. just 16.10% when enabled. Jitter decreased from 18.24 ms on average with ACRS4SDN disabled to 4.52 ms on average with ACRS4SDN enabled.

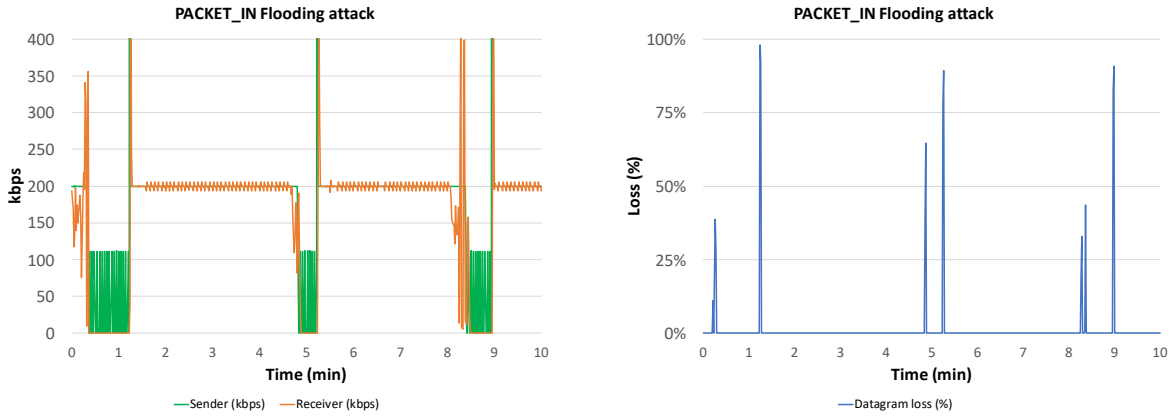


(a) (b)

Fig. 14—(a) Sender/Receiver throughput rates (kbps) vs. time, and (b) datagram loss (%) vs. time, for PACKET_IN flooding attack without response.

8.3 Scenario 3: Flow Rule flooding attack

The flow rule flooding attack was initiated by a malicious app in the SDN controller. The app submits a large number of spurious flows to a remote switch (i.e., a switch that is located across a bottleneck link from the SDN controller). These flows are immediately added to the SDN controller's flow table in the



(a) (b)

Fig. 15—(a) Sender/Receiver throughput rates (kbps) vs. time, and (b) datagram loss (%) vs. time, for PACKET_IN flooding attack with response.

PENDING_ADD state, and gradually transition to the ADDED state as they are transmitted via OpenFlow messages to the respective switch.

With ACRS4SDN disabled, the effects of this attack on the network are long-lasting. On average across 10 trials, the receiver throughput rate was 182.59 kbps, 8.46% of datagrams were lost, and jitter was 5.82 ms. See Figure 16 shows throughput rates and datagram loss during a flow rule flooding attack with ACRS4SDN disabled.

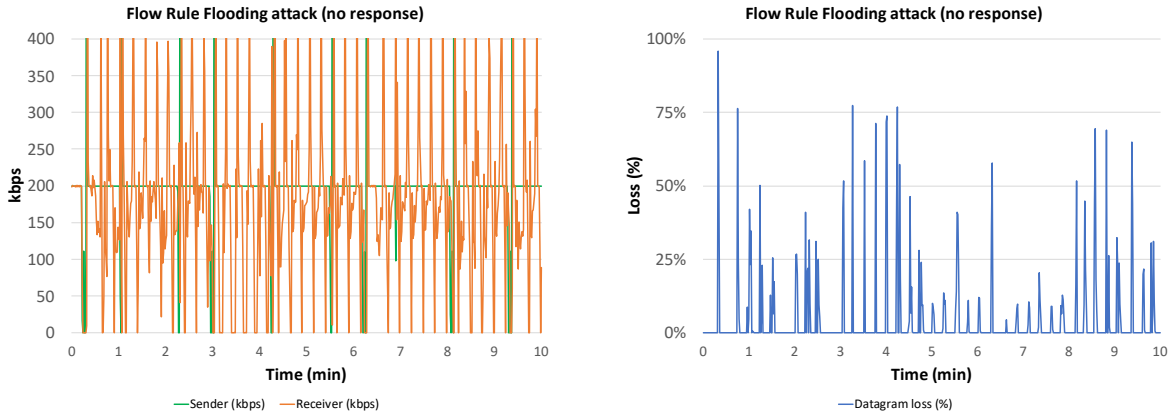
With ACRS4SDN enabled, the situation was much improved. Once detected, part of the response initiated by ACRS4SDN was to remove the malicious app. As such, the attack was relatively short-lived depending on how quickly it was detected. Generally, the faster it is detected, the fewer of the malicious flows actually make it out to the switches and thus the faster the cleanup process on the switches is.

See Figure 17 for sender and receiver throughput rates, as well as datagram loss percentage over time, during a flow rule flooding attack with ACRS4SDN enabled.

The average receiver throughput rate across 10 trials was 197.07 kbps, while the average datagram loss was 1.46%. Thus, ACRS4SDN was extremely effective at mitigating this attack.

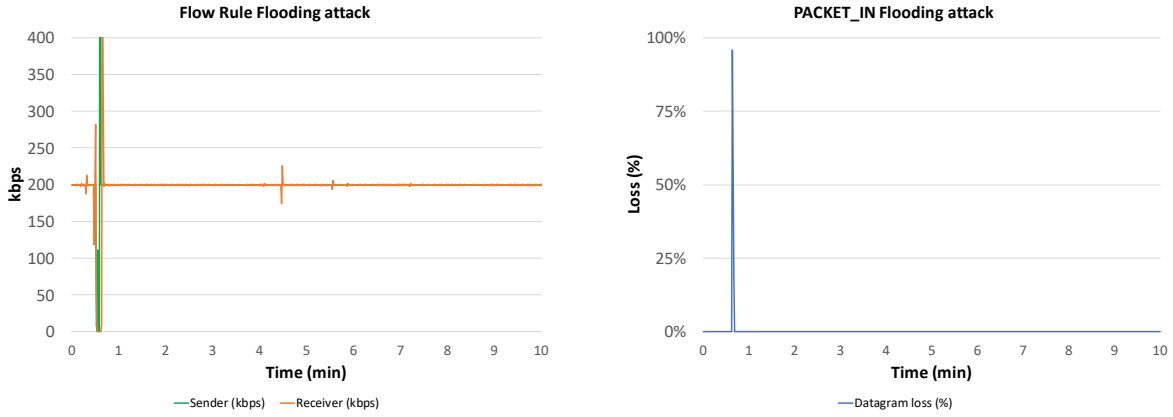
9. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced ACRS4SDN, a SDN system to enable autonomous cyber responses. We highlighted the properties of the cybersecurity domain and SDNs to motivate the use of refinement planning (RAE+UPOM) to accomplish automated attack recovery. The refinement methods of RAE+UPOM utilize expert domain knowledge to plan and execute recovery tasks for SDNs. Refinement methods are recovery procedures written by human experts, which can be complex algorithms 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 that apply. RAE+UPOM suggests to the Security



(a) (b)

Fig. 16—(a) Sender/Receiver throughput rates (kbps) vs. time, and (b) datagram loss (%) vs. time, for flow rule flooding attack without response.



(a) (b)

Fig. 17—(a) Sender/Receiver throughput rates (kbps) vs. time, and (b) datagram loss (%) vs. time, for flow rule flooding attack with response.

Manager of ACRS4SDN the best way to proceed. Our experiments show that ACRS4SDN successfully and effectively recovers a real SDN in multiple cyberattack scenarios.

Human domain experts, who write the refinement methods, need a good understanding of hierarchical operational models, in order to effectively express their knowledge of the cyber domain and tune the system to work well in practice. One area of future research is how to ease this burden through a methodology or templates that domain experts can use to encode their domain knowledge without the planning context, and how to expand the research results from ACRS4SDN to other IT systems.

Over time, cyber experts' knowledge is updated as new vulnerabilities and cyberattacks are discovered. The ACRS4SDN operational model should be updated as the knowledge of cyber experts advances. We are developing a scheme to update the operational model while ACRS4SDN is running without disruption of defense capabilities.

Sometimes it may be desirable to run ACRS4SDN in an advisory mode, rather than fully autonomous mode. In advisory mode, ACRS4SDN would receive tasks and plan actions as usual but provide the actions to a human in the loop, rather than executing the responses autonomously. In the future, we plan to implement the advisory mode and allow it to be selected at configuration time.

[6] showed that learning can be integrated with RAE+UPOM to improve the performance of UPOM by learning a heuristic function and by learning to choose refinement methods for tasks when there is not enough time to plan. The learning happens by generating training data with randomly generated tasks and contexts and feeding them to RAE+UPOM. Then, a neural network is trained to learn the "best" refinement method for a task and estimated utility values for different contexts. Future work will include integrating these learning strategies with ACRS4SDN and testing their performance. We believe that combining the strengths of both planning and learning is the best way to extend our work of automating attack recovery in dynamic environments like IT systems.

ACKNOWLEDGMENTS

The authors would like to acknowledge the Office of the Under Secretary of Defense for Research and Engineering (OUSD(R&E)) Cyber Research Program, and the Office of Naval Research (ONR), for their support for this work.

REFERENCES

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

2. L. Zhang, S. Yu, D. Wu, and P. Watters, "A Survey on Latest Botnet Attack and Defense," Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, 2011, pp. 53–60.

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

4. S. Yamaguchi, "Botnet Defense System: Concept and Basic Strategy," Proceedings of the 2020 IEEE International Conference on Consumer Electronics (ICCE), 2020, pp. 1–5.

5. S. Lee, J. Kim, S. Woo, C. Yoon, S. Scott-Hayward, V. Yegneswaran, P. Porras, and S. Shin, "A comprehensive security assessment framework for software-defined networks," *Computers and Security* **91**, 101720 (2020), ISSN 0167-4048, doi:https://doi.org/10.1016/j.cose.2020.101720. URL http://www.sciencedirect.com/science/article/pii/S0167404820300079.

6. S. Patra, J. Mason, A. Kumar, M. Ghallab, P. Traverso, and D. Nau, "Integrating Acting, Planning, and Learning in Hierarchical Operational Models," Proceedings of the Proceedings of the ICAPS International Conference on Automated Planning and Scheduling, 2020.

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

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

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

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

11. T. Y. Mu, A. Al-Fuqaha, K. Shuaib, F. M. Sallabi, and J. Qadir, "SDN Flow Entry Management Using Reinforcement Learning," *ACM Trans. Auton. Adapt. Syst.* **13**(2) (Nov. 2018), ISSN 1556-4665, doi:10.1145/3281032. URL `https://doi.org/10.1145/3281032`.

12. V. R, S. Kp, M. Alazab, S. Srinivasan, and S. Ketha, "A Comprehensive Tutorial and Survey of Applications of Deep Learning for Cyber Security," "" (01 2020), doi:10.36227/techrxiv.11473377.

13. D. S. Berman, A. L. Buczak, J. S. Chavis, and C. L. Corbett, "A Survey of Deep Learning Methods for Cyber Security," *Inf.* **10**, 122 (2019).

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

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

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

17. F. F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A high level supervision and control language for autonomous mobile robots," Proceedings of the ICRA, volume 1 (IEEE), 1996, pp. 43–49.

18. O. Despouys and F. Ingrand, "Propice-Plan: Toward a Unified Framework for Planning and Execution," Proceedings of the European Conference on Planning, 1999, pp. 280–292.

19. L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," Proceedings of the ecml, volume 6, 2006, pp. 282–293.

20. S. Patra, A. Velazquez, M. Kang, and D. Nau, "Using Online Planning and Acting to Recover from Cyberattacks on Software-defined Networks," *Proceedings of the AAAI Conference on Artificial Intelligence* **35**(17), 15377–15384 (May 2021). URL `https://ojs.aaai.org/index.php/AAAI/article/view/17806`.

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

22. D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL-the planning domain definition language," 1998.

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

This page intentionally left blank

Appendix A

UPOM: A UCT-LIKE SEARCH PROCEDURE

UPOM (UCT-like Procedure for Operational Models) performs a recursive search to find a method m for a task τ and a state s approximately optimal for a utility function U. It is a UCT-like [19] Monte Carlo tree search (MCTS) procedure over the space of refinement trees for τ (see Figure A3).

In games like Go, MCTS procedures have been used to analyze the most promising moves in the search tree by doing random sampling of paths in the tree. This is done by doing several playouts (called rollouts). In each such rollout, the game is played out to the very end by selecting moves at random until the game ends. The final result of each rollout is then used to weight the nodes in the search tree to guide the nodes that are to be chosen in future rollouts.

UPOM does Monte Carlo rollouts to search its search space of refinement trees. Each rollout consists of generating and evaluating a randomly-generated plan. At every task node that it encounters during a rollout, UPOM needs to choose an applicable method, and it makes this choice randomly using a statistical model of sequential experiments called the *multi-arm bandit problem*, which we will now explain.

In the multi-arm bandit problem, an agent i may choose an action from a set of actions $A = \{a_1, a_2, \ldots, a_k\}$. Each time i chooses action $a_j \in A$, i will receive a reward from a probability distribution p_j that is unknown but is assumed to be stationary. The objective is to choose a sequence of actions that maximizes the agent's expected payoff. Each time the agent chooses an action, it must choose between *exploitation* (choosing an action that has given high rewards in the past, in hopes of getting a high reward) and *exploration* (choosing an action that's less familiar, in hopes that it might produce an even higher reward). There is an algorithm called UCB1 [19] that maintains statistics on the number of times it has tried each action and the average reward received from those tries, and uses these statistics to choose its next action. More specifically, for each action $a \in A$, let

$$q(a) = \text{average reward agent } i \text{ has received from } a, \tag{A1}$$

$$n(a) = \text{number of times agent } i \text{ has tried } a, \tag{A2}$$

$$n_t = \sum_{a \in A} n(a), \text{ i.e., } i\text{'s total number of tries}, \tag{A3}$$

$$\phi(a) = q(a) + C\sqrt{(\ln n_t)/n(a)}. \tag{A4}$$

Then $\phi(a)$ is an estimate of a's desirability, that balances how desirable it would be to exploit a (based on the average reward received from a in the past) versus how desirable it would be to learn more about a (based on how many times a has been tried). The constant $C > 0$ determines whether to emphasize exploration (high C) or exploitation (low C).

Each time UCB1 is called, it chooses the action

$$\tilde{a} = \arg\max_{a \in A} \phi(a),$$

and updates the statistics based on the reward it receives from $\tilde{a}$. It can be shown that if we call UCB1 repeatedly for some number of times $k \to \infty$, then $\tilde{a}$ converges to the optimal action, i.e., the one with the highest expected reward.

UCT [19] is a well-known MCTS algorithm that uses UCB1 to search game trees and MDPs. Each time UCT is called, it does a Monte Carlo rollout, terminating the rollout at some cutoff depth d. At each state that it visits during the rollout, UCT uses UCB1 to choose the next action in the rollout. When the rollout ends, UCT uses a heuristic evaluation function to estimate the expected utility of the state that was reached at the end of the rollout, and uses that estimate to update UCB1's statistics for each state that UCT visited during the rollout. If one calls UCT repeatedly at some state s for some number of times $k \to \infty$, the action a^* chosen by the k'th call to UCT converges to the action having the highest estimated expected utility.

UPOM also does Monte Carlo rollouts, but it has a more complicated search space and it chooses among refinement methods instead of actions.[A.1] At each task node τ that UPOM encounters during its Monte Carlo rollouts, the objective is to choose a method that will optimize a utility value of the resulting plan, e.g., to minimize expected cost, or to maximize expected cost-effectiveness (a utility value that we'll explain later). UPOM keeps UCB1-like statistics on the set of methods $M_{s,\tau}$ that are applicable for τ in the current state s, and uses these statistics to choose a method $\tilde{m} \in M_{s,\tau}$ in a similar manner to UCB1's choice among actions. The statistics are as follows, where q and n are as in Equations A1 and A2:

- $Q(\tau) = [q(m) \mid m \in M_{s,\tau}]$, i.e., a list of the $q(m)$ values of the available methods for τ that are applicable in the current state,
- $N(\tau) = [n(m) \mid m \in M_{s,\tau}]$, i.e., a list of the $n(m)$ values of the available methods for τ that are applicable in the current state.

From these statistics, UPOM computes a value $\phi(m)$ for each $m \in M_{s,\tau}$ using a formula similar to Equation A4. As the next method to use in the rollout, UPOM chooses the one that has the best ϕ value (i.e., the maximum or minimum value, depending on the utility function to be optimized). At the end of each Monte Carlo rollout, UPOM evaluates the plan produced by the rollout, and uses its utility value to update the Q and N values at each task node encountered during the rollout.

Let us see an example of how UPOM can be used to refine the task τ in Figure A3 and how the statistics are updated after each rollout. Since the task τ has two applicable refinement methods, m and m', UPOM will first explore both at least once. Figure A1 informally shows how the values of Q (estimated utilities) and N (number of visits) counts are updated at each task node of the search tree, one rollout at a time, for the first three rollouts. For simplicity, let us assume that UPOM is minimizing cost. Say, in the first rollout (first call to UPOM), m is chosen to refine τ and the cost of the rollout is found to be 10. The Q-values of the corresponding visited nodes are updated to be 10. In the second rollout, since m' hasn't been explored yet, UPOM will choose m' to refine τ. The cost of the rollout is computed to be 16, and the Q and N values are updated accordingly after the rollout is done. For the third rollout, UPOM chooses m using the UCB1 formula and computes the cost to be 12. After three rollouts, the Q-values vector for τ is $[11, 16]$ suggesting that the expected cost for m is 11 and for m' is 16. So, UPOM will suggest m' since the objective is to minimize cost.

[A.1] UPOM needs to choose among alternative methods, but not alternative actions. When RAE is executing a refinement method m, it performs the actions that m says to perform; and when UPOM simulates the execution of m, it simulates the performance of the actions that m says to use.

	Task node	τ	τ_1	τ_2
Initialization	Q	[-,-]	[-,-]	[-,-]
	N	[0,0]	[0,0]	[0,0]
Rollout 1	Q	[10,-]	[10,-]	[10,-]
Cost = 10	N	[1,0]	[1,0]	[1,0]
Rollout 2	Q	[10,16]	[10,-]	[10,-]
Cost = 16	N	[1,1]	[1,0]	[1,0]
Rollout 3	Q	[11,16]	[10,12]	[10,12]
Cost = 12	N	[2,1]	[1,1]	[1,1]

Fig. A1—A table informally showing how the Q-values and N counts are updated for every task/sub-task after each rollout (a call to UPOM) for the search tree shown in Figure A3.

To choose what method to use for some task τ, RAE can call UPOM repeatedly for some number of times. It can interrupt these calls at any time, which is essential for a reactive actor in a dynamic environment. When it does so, it can then choose the method $\tilde{m}$ having the best value for $q(\tilde{m})$, which is an estimate of m's expected utility.

Provided that some restrictions are satisfied, it is possible to show that if UPOM is called repeatedly for some task τ and state s, then as the number k of calls to UPOM goes to ∞, the method chosen by the k'th call converges to the method having the optimal expected utility.

Acting problem. The deliberative acting problem for ACRS4SDN can be stated informally as follows: given Σ 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) τ in a current state s. For the example of a PACKET_IN flooding attack, this reduces to choosing one among the three possible candidates as shown in Figure A2. ACRS4SDN 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.

[!b]

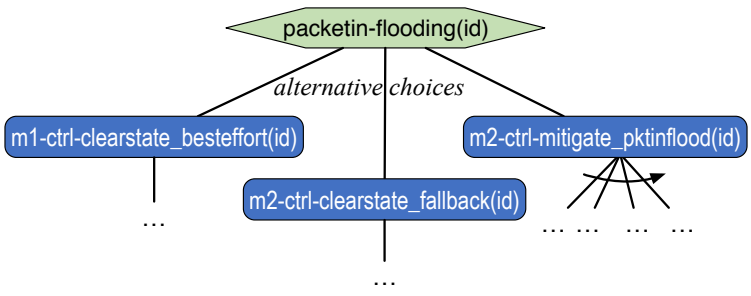


Fig. A2—Part of the space of refinement trees for a PACKET_IN flooding attack. The three alternative choices are the refinement methods shown in Figures 3, 4, and 5.

The current context for an incoming attack τ_0 is represented via a *refinement stack* σ, which keeps track of how much further RAE has progressed in recovering from τ_0. The refinement stack is a LIFO list of tuples $\sigma = \langle(\tau, m, i), \ldots, (\tau_0, m_0, i_0)\rangle$, where τ is the deepest current sub-task in the refinement of τ_0, m is the method used to recover from τ, i is the current instruction in $body(m)$, σ is handled with the usual stack push, pop and top functions.

When RAE addresses a task or event τ, it must choose a method m to handle τ. Purely reactive RAE makes this choice arbitrarily; more informed RAE relies on a planner. Once a method 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 σ 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.

In summary, RAE follows a refinement tree as in Figure A3. At an action node it performs the action on the SDN; if successful it pursues the next step of the current method, or higher up if it was its last step; if the action fails, an alternate method is tried. This goes on until the SDN has recovered from the attack, or until no alternate method is available in the current state. 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.

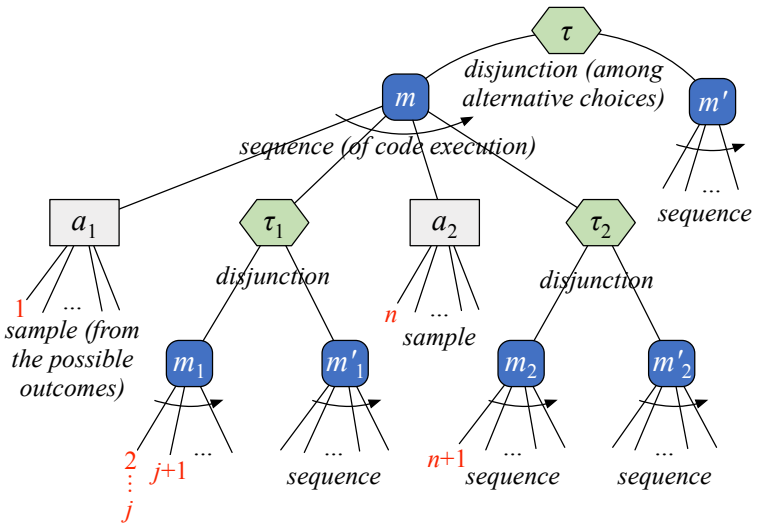


Fig. A3—The space of refinement trees for a task τ. A *disjunction* node is a task followed by its applicable methods. A *sequence* node is a method m followed by all the steps. A *sampling* node for an action a has the possible nondeterministic outcomes of a as its children. An example of a Monte Carlo rollout in this refinement tree is the sequence of nodes marked 1 (a sample of a_1), 2 (first step of m_1), ..., j (subsequent refinements), $j+1$ (next step of m_1), ..., n (a sample of a_2), $n+1$ (first step of m_2), etc.