3-2021

# Strategic Upgrade Scheduling and Portfolio Management to Fortify Contested Networks

Justin L. Anderson


# Strategic Upgrade Scheduling and Portfolio Management to Fortify Contested Networks

THESIS

Justin L. Anderson, Capt, USAF

AFIT-ENS-MS-21-M-141

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

***AIR FORCE INSTITUTE OF TECHNOLOGY***

**Wright-Patterson Air Force Base, Ohio**

AFIT-ENS-MS-21-M-141

# STRATEGIC UPGRADE SCHEDULING AND PORTFOLIO MANAGEMENT TO FORTIFY CONTESTED NETWORKS

THESIS

Presented to the Faculty

Department of Operational Sciences

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Operations Research

Justin L. Anderson, BS

Capt, USAF

March 25, 2021

AFIT-ENS-MS-21-M-141

# STRATEGIC UPGRADE SCHEDULING AND PORTFOLIO MANAGEMENT TO FORTIFY CONTESTED NETWORKS

THESIS

Justin L. Anderson, BS
Capt, USAF

Committee Membership:

Lt Col Timothy W. Holzmann, PhD
Chair

Dr. Brian J. Lunday
Member

AFIT-ENS-MS-21-M-141

# Abstract

The nuclear command, control, and communication (NC3) network provides an effective deterrent against adversaries by maintaining robust communication between assets, operators, and decision makers. NC3 must receive a series of upgrades to remain an effective deterrent against evolving threats around the globe. Given a contested network and a portfolio of network upgrades, this research seeks an optimal upgrade schedule to maximize network resilience in support of NC3 modernization. Our research contributes a model that solves a budget-constrained network upgrade scheduling problem (NUSP). The NUSP is a defender-attacker-defender tri-level optimization model that achieves fortification via selecting and scheduling network upgrades. The lower-level defender problem is the shortest path problem, the mid-level attacker problem is the shortest path interdiction problem (SPIP), and the upper-level defender's problem is the NUSP. We present an exact, solvable solution method and propose a simulated annealing (SA) heuristic. The formulation is applied to a large network instance. Using four scenarios, we investigate the effects of attacker budget and risk profiles on the optimal upgrade schedule. Results indicate that both attacker budget and risk profile affect the optimal upgrade strategy. Moreover, the optimal upgrade schedule provides greater network resilience than the myopic (i.e., baseline) upgrade schedule.

# Acknowledgements

I would like to express my sincerest gratitude to my advisor, Lt Col Timothy Holzmann, for his exuberant leadership and unwavering support. His expertise and knowledge proved invaluable during my research. I also wish to thank Dr. Brian Lunday for his keen insight and uplifting encouragement. He always provided me a clear path forward.

Justin L. Anderson

# Table of Contents

Page

# List of Figures

# List of Tables

STRATEGIC UPGRADE SCHEDULING AND PORTFOLIO MANAGEMENT TO FORTIFY CONTESTED NETWORKS

# I. Introduction

## 1.1 Motivation and Background

Nuclear command, control, and communication (NC3) systems perform a vital role in guarding national security by maintaining robust communication flow between sensors, authorized decision makers, and weapon system operators. This begins with the detection of an incoming attack, followed by the flow of threat information to the decision authority, and finally the response directive must flow from the decision authority to the assets required for a successful counterstrike. A robust NC3 network provides an effective deterrent against potential adversaries by ensuring that any provocative action is met with a devastating response. In addition to deterrence, NC3 also provides assurance to United States (US) allies by extending an umbrella of protection to countries that seek refuge from common enemies. Unfortunately, the present nuclear infrastructure is outdated and must be upgraded to preserve its effectiveness.

According to Deptula et al. (2019), the NC3 architecture received its last major upgrade in the 1980s during the peak of the Cold War. Gen John Hyten, commander of US Strategic Command (USSTRATCOM), stated in March 2017 that the aging NC3 system is his biggest concern, and modernizing NC3 is his number one priority. The Secretary of Defense (SecDef) released the Nuclear Posture Review in 2018 that addressed the need for phased modernization of NC3 to withstand the evolving threats

around the globe (Office of the Secretary of Defense, 2018). Guided by the SecDef and Secretary of the Air Force (SecAF), the Strategic Deterrence and Nuclear Integration Office (AF/A10) expressed the need for improved command and control (C2) node capability and communication path connectedness to increase the resilience of the NC3 network.

In modernizing the NC3 network, USSTRATCOM seeks to deter an adversary from attacking the NC3 network. In this thesis, we define an attack as a communication delay that jeopardizes time-sensitive decision-making processes and restricts the ability to respond to strategic threats. From the Deterrence Operations Joint Operating Concept, we deter adversarial activities either by imposing cost of action or by denying the benefit of action (Department of Defense, 2006). For example, we deny benefit of an attack on NC3 through redundant communication pathways so that attacks do not achieve the intended level of communications delay. Alternatively, we might impose cost to the attacker by fortifying the network. If an adversary must expend an unacceptable amount of resources in order to achieve a minimum level of communication delay, then the adversary is deterred from launching the attack.

The current nuclear network maintains centralized C2 nodes and communication paths across a multi-domain network (Deptula et al., 2019). If necessary, the network resorts to backup nodes and communication paths to maintain communication flow (Deptula et al., 2019). These redundant systems increase network resilience to ensure communication between assets. Walker et al. (2003) define network resilience as "the capacity of a system to absorb disturbance and reorganize while undergoing change so as to still retain essentially the same function." In the same way, the NC3 network must handle attacks, also referred to as interdictions, and preserve communication flow to ensure a successful mission. With respect to NC3, network resilience is defined as the persistence and preservation of communication flow through the net-

work despite network interdiction. We measure network resilience by determining the quickest communication path (i.e., the shortest path) between the source node and terminus node *after* an interdiction occurs.

To design a resilient NC3 network, decision makers must prioritize and select a series of upgrades that either update or replace portions of the existing NC3 network. However, it may be unclear how an upgrade improves the overall resilience of the network. Additionally, these upgrades come at a cost and often require acquisition lead time before reaching initial operational capability (IOC). For instance, a relatively quick update to a predetermined network component may provide marginal increase to resilience at a relatively low cost and reach IOC within two years. On the other hand, a competing upgrade might replace that same network component with better resilience but cost five times more and take 10 years to reach IOC. Selecting between these competing upgrades is not always easy. Moreover, a realistic upgrade portfolio will contain substantially more upgrade options; therefore, the solution space becomes exponentially more complex. These complexities warrant a framework that assists decision makers with prioritizing upgrades and organizing them in a fiscally responsible upgrade schedule.

## 1.2 Problem Statement

Given a contested network and a portfolio of network upgrades, this research seeks an optimal upgrade schedule to maximize network resilience in support of NC3 modernization. A fundamental concept used to solve this problem is the two-player Stackelberg game (von Stackelberg, 1952). The following explanation describes the Stackelberg game within the context of the NC3 problem. The first player is the defender who seeks to minimize communication time across the network. The second player is the attacker who seeks to maximize the defender's communication time

by strategically interdicting the network. The defender responds to the attacker by upgrading the network and fortifying it against future attacks from the second player. After the defender selects a fortification strategy, the second player observes the changes to the network and chooses an interdiction strategy. The attacker's objective is to disrupt or deny the flow of communication over the network by maximizing the delay of communication.

A Stackelberg game assumes that players alternate turns in sequence, and both players act on perfect information. The assumption that the attacker has full information of the network is conservative and presents a worst-case scenario from the defender's perspective. Furthermore, a zero-sum Stackelberg implies that both players act on the same objective function, and one player's loss is directly proportional to the other player's gain.

In alignment with AF/A10's primary concerns (i.e., communication delay), we choose a modeling construct that enables the attacker to delay communication on arcs within the network. The attacker's goal of finding a maximal interdiction is modeled using the shortest path interdiction problem (SPIP) (Israeli and Wood, 2002). In the SPIP, the interdictor's objective is to maximize the communication delay of the network by interdicting arcs (i.e., communication paths) required for quick communication. When an arc is interdicted, a delay is imposed to that arc. Building on the SPIP, we model the defender's fortification decision by scheduling network upgrades (i.e., the addition of nodes and arcs). Both the attacker's interdiction strategy and the defender's upgrade schedule are subject to respective budgetary constraints. This problem assists the defender in selecting a fortification strategy with the objective of mitigating an attacker's most damaging interdiction strategy. Unlike other research, this fortification strategy involves selecting and scheduling network upgrades.

The complexities of our network fortification problem are exacerbated by the

multi-modal nature of communication pathways in the NC3 network. For instance, the communication between computer servers performs differently than communication between satellites. The functionality differences often result in distinct performance traits. Our research handles these performance traits by attributing each arc with its own performance measure. The NC3 network is accurately characterized by defining the performance attributes of each arc.

## 1.3 Research Objectives

Our research contributes a model that solves a budget-constrained network upgrade scheduling problem (NUSP) and achieves fortification via selecting and scheduling network upgrades. Our model improves the NC3 upgrade acquisition process by simplifying programmatic level decisions to maximize NC3 resilience. The specific research objectives include developing an exact formulation to solve the tri-level NUSP, comparing network resilience improvement between the optimal solution and a myopic solution, and evaluating the effects of the defender's risk preferences and attacker's budget on the optimal upgrade schedule.

## 1.4 Organization of the Thesis

The organization of this thesis is as follows: Chapter II provides a review of research relating to network interdiction, network fortification, vulnerability assessment, and scheduling problems. Chapter III outlines the tri-level formulation that solves the NUSP and presents an exact solution algorithm and a heuristic algorithm. Chapter IV defines an unclassified transportation network for testing purposes, analyzes the solution results, and investigates the solution's sensitivity to varying risk preferences and attacker budgets. Lastly, Chapter V concludes the thesis and proposes additional topics for future research.

# II. Literature Review

## 2.1 Overview

A growing importance is placed on network resilience in support of supply chains, public transportation, and military defense. To better understand the intricacies of improving network resilience, we discuss the types of network disruptions, approaches to solve network interdiction problems, fortification strategies, and vulnerability assessments. Finally, we present a review of scheduling problems to inform network upgrade scheduling.

## 2.2 Background

A network disruption is a negative impact to network operations that causes a reduction in effectiveness. In the context of NC3, a network disruption interrupts communication flow causing an increase in communication delay. Network disruptions fall under one of two categories, random and targeted (Kang et al., 2011). A random disruption occurs to any network component without regard for the component's purpose or contribution to the network. Examples of random disruptions include natural disasters such as tornadoes and earthquakes. A random disruption may be modeled such that the likelihood of disruption is equal for all network components. A targeted disruption consists of a premeditated attack to the network with specific objectives. Examples of targeted disruptions include terrorist attacks and blockades. Critical components, such as a main thoroughfare or supply hub, are more often targeted since intelligent attackers are responsible for the attack. Kang et al. (2011) seek a balance between random and targeted disruption resilience in a supply chain network.

There are two primary methods of modeling network disruptions and network in-

terdiction problems. First, stochastic programming (SP) utilizes probabilistic events and optimizes over the expected outcome given the likelihood of all possible events. Stochastic models are useful when the probability of each event is known, and the goal is to optimize the solution's expected value. For instance, SP may be useful when modeling random disruptions. SP incorporates the probability of each random disruption, and the resulting model is optimized for multiple random disruptions rather than a single, worst-case disruption. Second, a robust optimization (RO) approach may be used to optimally solve a worst-case disruption. This approach is best suited for reducing the impact of the most severe disruption. Maggioni et al. (2017) compare SP and RO and discuss the advantages and disadvantages of each. Our research utilizes RO due to the severity of consequences (i.e., mission failure) of a worst-case attack on the NC3 network.

The RO approach leads us to the theory of Stackelberg games (von Stackelberg, 1952). These are two-player games where the players alternate moves, and each player acts with perfect knowledge of the other player's moves. The assumption of perfect knowledge translates directly to the robust assumption since the attacker, having full knowledge of the network, may readily identify the most damaging interdiction strategy (Smith et al., 2013). In most network interdiction formulations the attacker and defender are playing a zero-sum game with both players seeking to optimize the same objective, albeit in opposite directions (Smith et al., 2013). The turn-based play of Stackelberg games also matches the sequence of decisions in our network interdiction problem. These turns are frequently modeled in the literature using multi-level mathematical programs.

## 2.3 Network Interdiction

Researchers often use bi-level mathematical programs to model game play of competitors in a two-player attacker-defender (AD) Stackelberg game. The seminal work of Wood (1993) presents a bi-level mixed-integer program for deterministic network interdiction. Israeli and Wood (2002) investigate improvements in computation efficiency to solve the SPIP using the Benders decomposition algorithm. However, the Benders decomposition algorithm is sensitive to high interdiction delay values; run time increases as the magnitude of delays increase. Israeli and Wood (2002) present a solution to this inefficiency with a covering decomposition algorithm. The covering decomposition problem is especially useful when the attacker is capable of severing arcs (i.e., a very large delay resulting in an impassable arc). Lozano and Smith (2017) also solve the SPIP but improve upon the computational efficiency of the decomposition algorithms presented by Israeli and Wood (2002) with a backward sampling framework.

Morton et al. (2007) introduce SP to their bi-level interdiction models. Their interdiction models seek the reduction of nuclear smuggling. The smuggler seeks to maximize the probability of evading detection, whereas the interdictor's problem minimizes the smuggler's probability of evading detection. This research is especially interesting as it relates to a stochastic origin and destination of smuggler activities.

## 2.4 Network Fortification

The SPIP is extended to a tri-level optimization problem to allow the defender an opportunity to fortify the network in expectation of the attacker's interdiction activity. This fortification problem takes a defender-attacker-defender (DAD) construct. Lazzaro (2016) uses tri-level optimization to solve a DAD problem. Specifically, the defender's fortification protects or reinforces network components to increase its

resilience against interdiction. Lazzaro (2016) proposes a computationally quicker solution to find nested defenses when the defender and attacker budgets are unknown.

Ghorbani-Renani et al. (2020) discuss a tri-level optimization formulation that minimizes the network's recovery trajectory over time. This article is particularly useful because it demonstrates a time-sensitive approach to fortify the network by minimizing the cumulative unmet demand over time. Ghorbani-Renani et al. (2020) use the covering decomposition method outlined by Israeli and Wood (2002) to solve the AD interdiction problem. Their results indicate that it is beneficial to simultaneously solve for both pre-disruption reinforcement strategies and post-disruption crew scheduling and resource assignments in order to recover disrupted network components in a timely manner.

### 2.5 Vulnerability Assessment

A common approach to assess network vulnerability is to identify the most critical network components. Cantillo et al. (2019) propose a transportation network vulnerability assessment that identifies critical arcs. Cantillo et al. (2019) prioritize the rehabilitation of critical arcs to inform planners and decision makers of ways to improve natural disaster response. Du et al. (2014) provide a vulnerability scanning algorithm to categorize the most critical network components, but they must use a heuristic to scale their methods for large networks. Additionally, Du et al. (2014) find that even though arc disruptions are more commonly evaluated, node disruptions may better characterize widespread disaster. For this reason, Du et al. (2014) use the vulnerability scanning algorithm to assess both arcs and nodes within the network.

Researchers have found that identifying critical network components is a non-trivial task. For example, Alderson et al. (2013) find that a rule-of-thumb policy is insufficient when identifying critical network components. Similarly, Brown et al.

(2006) conclude that heuristic approaches to identify network vulnerabilities are inadequate. Brown et al. (2006) demonstrate how their formulation identifies essential network components within an electric grid network that requires protection. The essential network components are identified based on the attacker's optimal interdiction strategy. Brown et al. (2006) highlight how changes to the attacker's budget may lead to completely different interdiction strategies, which subsequently may adjust the defender's optimal fortification strategy. Not only are these interdiction strategies different, but Brown et al. (2006) state "[they] may have nothing in common." This is a significant insight for DAD formulations when the attacker is constrained to a budget.

Zhao et al. (2019) suggest a different approach to assess network vulnerability. They focus on evaluating the topology of a supply chain network as an indicator of robustness where the objective is to maximize flow across the network. Zhao et al.'s (2019) methodology removes one arc at a time (i.e., disruption) and then re-evaluates the network after each removal. The robustness is measured as a function of throughput in the supply chain. This methodology is tested with a variety of disruptive scenarios.

## 2.6 Scheduling

Although determining the desired network end-state is an important goal for maximizing network resilience, it is prudent to generate a schedule that outlines a series of upgrades to reach the desired network end-state. Often these network upgrades span years or even decades due to budget constraints and lead time before upgrades become operational. Chow et al. (2011) discuss the need for a flexible planning approach to a transportation network that can be applied to a variety of network problems such as the traveling salesman problem or transshipment problem. They discuss the importance of modeling a multi-period transportation network design that enables a flexible

and adaptable planning approach due to the insufficiency of static planning. To accomplish this task, the current network state and all potential network improvements must be known. All potential improvements require an approximated cost and known performance attributes (e.g., when modeling the shortest path problem, the cost to build an arc and the arc length must be known). This research supports planning agencies by providing a framework that enables adaptive network redesign.

Garey et al. (1976) develop a flowshop scheduling problem and prove that finding the shortest length schedule is an NP-complete problem. They constrain their problem by only allowing one job per machine. Each job is a sequence of tasks, and once a task starts processing, it must finish. Furthermore, they define a schedule as the time each task starts processing for a given set of jobs. The authors conclude that a heuristic is the best approach to return near-optimal solutions for large scheduling problems. Van Laarhoven et al. (1992) use a heuristic to solve the job shop scheduling problem. Their results reveal that the simulated annealing (SA) algorithm, when compared to the classical iterative improvement approach, increases computational efficiency and returns high quality solutions. The gap in computational efficiency increases as the problem size increases (Van Laarhoven et al., 1992).

# III. Solution Methodology

## 3.1 Background

The network upgrade scheduling problem (NUSP) is solved as a tri-level optimization mathematical model. The tri-level optimization model takes the defender-attacker-defender construct where the lower-level defender problem minimizes the shortest path. The mid-level attacker problem maximizes the shortest path via the shortest path interdiction problem (SPIP). Finally, the upper-level defender problem minimizes the SPIP by selecting, prioritizing, and scheduling network upgrades. We call a solution to the NUSP an acquisition strategy. The optimal acquisition strategy reduces the impact of the attacker's most damaging attack strategies. Furthermore, the optimal solution to the mid-level SPIP informs the defender of potential vulnerabilities in the network at each stage of the upgrade schedule by identifying specific interdiction strategies of the attacker. This chapter discusses how to formulate the tri-level NUSP, introduces an exact, solvable solution method, and proposes a simulated annealing (SA) heuristic.

## 3.2 Tri-level Optimization Problem

### 3.2.1 Overview

The problem inputs are presented in order of each level in the tri-level formulation to solve the NUSP. The inputs for the lower-level shortest path problem are introduced first, followed by the mid-level attacker's SPIP, and finally the upper-level defender's upgrade scheduling problem.

### 3.2.2 Shortest Path Problem (lower-level)

To solve the shortest path problem, we initialize the network with a graph $G$ given a set of nodes $N$ and a set of arcs $A$. We identify a source node and a terminus node where the flow across network $G$ starts and ends. The net-flow across each node $n \in N$ is denoted as $f_n$. Therefore, we set $f_{source}$ to $-1$ to signify the initial flow out of the source node. Similarly, $f_{terminus}$ is set to 1 to signify where the flow ends. For all other nodes, $f_n$ is set to 0. The shortest path is determined by finding the shortest length path in the network between the source and terminus nodes. Therefore, the arc length $c_{ij}$ must be determined for every arc $(i, j) \in A$. The shortest path problem assumes that all arc lengths are non-negative to eliminate cycles. Finally, the decision variable $y_{ijt}$ indicates the defender's decision to traverse an arc.

| | |
|---|---|
| $n \in N$ | Set of all nodes in the network |
| $(i, j) \in A$ | Set of all arcs in the network |
| $t \in T$ | Set of time periods, $\{1, 2, ..., \lvert T \rvert\}$ |
| $f_n$ | Net flow across node $n$ where |

$$f_n = \begin{cases} -1 & n = \text{source} \\ 1 & n = \text{terminus} \\ 0 & \text{otherwise} \end{cases}$$

| | |
|---|---|
| $c_{ij}$ | Length of arc $(i, j)$, $\mathbb{R}_+$ |
| $y_{ijt} \in \{0, 1\}$ | Indicates if arc $(i, j)$ is traversed by defender at time $t$ |

### 3.2.3 Shortest Path Interdiction Problem (mid-level)

The mid-level SPIP contains its own set of variables. The attacker is constrained to a budget $b^{\text{atk}}$. When the attacker interdicts arc $(i, j)$ with a determined number

of resources $b$, a delay occurs corresponding to the value within the delay matrix $d_{ijb}$, where $b$ is the incremental value between 1 and $b^{\text{atk}}$. Table 1 provides an example of this delay matrix. Typically, as the number of resources used for interdiction increases, so does the delay to the arc. Therefore, the delay values in each row of the matrix are sorted in non-decreasing order. Finally, the binary decision variable $x_{ijbt}$ indicates the attacker's decision to interdict arc $(i, j)$.

| | |
|---|---|
| $b^{\text{atk}}$ | Attacker's budget, $\mathbb{Z}_+$ |
| $b \in B$ | Set of attacker's resources for interdiction, $\{1, 2, ..., b^{\text{atk}}\}$ |
| $d_{ijb}$ | Delay imposed on arc $(i, j)$ by spending resources $b$, $\mathbb{R}_+$ |
| $x_{ijbt}$ | Binary; 1 if arc $(i, j)$ is attacked at time $t$ with resources $b$, else 0 |

**Table 1. Delay Matrix Example**

| | Resources ($b$) | | | |
|---|---|---|---|---|
| Arc $(i, j)$ | 1 | 2 | 3 | 4 |
| (1,2) | 0 | 1 | 1 | 2 |
| (2,3) | 1 | 2 | 4 | 5 |
| (1,3) | 0 | 0 | 3 | 4 |

### 3.2.4 Scheduling Problem (upper-level)

The scheduling problem requires a set of upgrades $U$. For each upgrade $u \in U$, we let $O_u$ represent the set of nodes and arcs that are added to the network if upgrade $u$ is selected. We assume an upgrade goes through two phases, acquisition and operations. During the acquisition phase, an upgrade is under development and its network components $O_u$ are not usable. The operations phase signifies when upgraded network components $O_u$ become usable which means the network components may be traversed and interdicted. To distinguish between phases, $p = 1$ represents the acquisition phase and $p = 2$ represents the operations phase. Each upgrade $u \in U$ has

an associated lead time $\ell_u$, periodic acquisition cost $m_{up}$ where $p = 1$, and periodic operations cost $m_{up}$ where $p = 2$. This problem assumes there are no gaps between phases and that phases cannot overlap in the same time period. Furthermore, this model assumes an upgrade must complete acquisition before operations, and the number of time periods to complete acquisition must be met.

| | |
|---|---|
| $u \in U$ | Set of upgrades |
| $p \in P$ | Set of phases, acquisition ($p = 1$) and operations ($p = 2$) |
| $O_u \in A$ | Set of upgraded arcs implemented when upgrade $u$ is operational |
| $m_{up}$ | Defender's monetary cost for upgrade $u$ during phase $p$, $\mathbb{R}_+$ |
| $\ell_u$ | Lead time for upgrade $u$ to complete acquisition, $\mathbb{Z}_{++}$ |
| $b_{pt}^{\text{def}}$ | Defender's budget for phase $p$ at time $t$, $\mathbb{R}_+$ |
| $r_t$ | Risk of attack at time $t$, $\mathbb{R}_+$ |
| $w_{upt} \in \{0, 1\}$ | Binary; 1 if upgrade $u$ is in phase $p$ at time $t$, else 0 |

The upper-level scheduling problem also requires the defender's budget $b_{pt}^{\text{def}}$ for each phase $p \in P$ and each time period $t \in T$. The perceived risk of an attack occurring $r_t$ must be defined for each time period $t \in T$. Understanding the risk of an attack is especially important when selecting between a short-term risk-averse schedule versus a long-term risk-averse schedule. Finally, the binary decision variable $w_{upt}$ indicates the defender's decision to select an upgrade.

### 3.2.5 Formulation

$$z_{\text{NUSP}} = \min_{w} \; z_{\text{SPIP}(w)} \tag{1a}$$

$$\text{s.t.} \sum_{p \in P} w_{upt} \leq 1, \; \forall \; u \in U, \; \forall \; t \in T, \tag{1b}$$

$$m_{up} \cdot w_{upt} \leq b^{\text{def}}_{pt}, \; \forall \; u \in U, \; \forall \; p \in P, \; \forall \; t \in T, \tag{1c}$$

$$\ell_u \cdot w_{up+1t} \leq \sum_{t'=1}^{t} w_{upt'}, \; \forall \; u \in U, \; \forall \; p \in P \setminus \{2\}, \; \forall \; t \in T, \tag{1d}$$

$$\sum_{t \in T} w_{upt} \leq \ell_u, \; \forall \; u \in U, \; \forall \; p \in P \setminus \{2\}, \tag{1e}$$

$$\sum_{p \in P} w_{upt} \geq \sum_{p \in P} w_{upt-1}, \; \forall \; u \in U, \; \forall \; p \in P, \; \forall \; t \in T \setminus \{1\}, \tag{1f}$$

$$w_{upt} \in \{0, 1\}, \; \forall \; u \in U, \; \forall \; p \in P, \; \forall \; t \in T, \tag{1g}$$

$$\text{where} \quad z_{\text{SPIP}(w)} = \max_{x} \; z_{\text{SPP}(x,w)} \tag{2a}$$

$$\text{s.t.} \sum_{b \in B} x_{ijbt} \leq 1, \; \forall \; (i,j) \in A, \; \forall \; t \in T, \tag{2b}$$

$$\sum_{(i,j) \in A} \sum_{b \in B} b \cdot x_{ijbt} \leq b^{\text{atk}}, \; \forall \; t \in T, \tag{2c}$$

$$x_{ijbt} \in \{0, 1\}, \; \forall \; (i,j) \in A, \; \forall \; b \in B, \; \forall \; t \in T, \tag{2d}$$

$$\text{where} \quad z_{\text{SPP}(x,w)} = \min_{y} \sum_{(i,j) \in A} \sum_{b \in B} \sum_{t \in T} r_t \; (c_{ij} + d_{ijb} \cdot x_{ijbt}) \; y_{ijt} \tag{3a}$$

$$\text{s.t.} \sum_{j:(i,j) \in A} y_{ijt} - \sum_{j:(j,i) \in A} y_{jit} \geq f_i, \; \forall \; i \in N, \; \forall \; t \in T, \tag{3b}$$

$$y_{ijt} \leq w_{u2t}, \; \forall \; (i,j) \in O_u, \; \forall \; u \in U, \; \forall \; t \in T, \tag{3c}$$

$$y_{ijt} \geq 0, \; \forall \; (i,j) \in A, \; \forall \; t \in T. \tag{3d}$$

The objective function for the upper-level scheduling problem minimizes the attacker's SPIP by upgrading the network (1a). The programmatic constraints for the upper-level scheduling problem prohibit an upgrade from being in multiple phases

at the same time (1b), limit acquisition spending to the defender's budget for each phase (1c), and require the acquisition phase to end before starting the operations phase (1d). Additionally, the programmatic constraints bound the number the acquisition phase time periods (1e), and prohibit an upgrade from containing gaps between phases (1f). The upper-level scheduling problem also defines the binary constraint to indicate the acquisition phase of an upgrade (1g).

The objective function for the mid-level SPIP maximizes the shortest path via network interdiction (2a). The SPIP constraints force the selection of a single interdiction strategy for each arc (2b) and bound the attackers budget (2c). Constraint (2d) defines the binary variable that indicates if the attacker interdicts an arc.

Lastly, the objective function for the lower-level shortest path problem determines the shortest path in the network (3a). The shortest path problem constraints manage the conservation of flow between the source node and the terminus node (3b). Constraint (3c) prohibits non-operational arcs from being traversed by the defender. Constraint (3d) defines the non-negative decision variable that indicates the defender's shortest path.

## 3.3 Exact, Solvable Problem

### 3.3.1 Overview

To our knowledge, no existing solver engines can solve a tri-level formulation such as (1a) – (3d). In this section, we propose an exact algorithm that solves (1a) – (3d) in two steps. Step 1 enumerates every interdiction strategy by solving the SPIP($\boldsymbol{s}$) (i.e., (2a) – (3d)) for every upgrade state configuration $\boldsymbol{s} \in S$. Once the interdiction strategy is calculated for every $\boldsymbol{s} \in S$, the objective value of each upgrade state configuration $v_{\boldsymbol{s}}$ is passed as a parameter to step 2. Step 2 seeks an optimal ordering of upgrade state configurations. Sections 3.3.2 and 3.3.3 discuss the exact approaches

used to solve Step 1 and 2.

### 3.3.2 Step 1: Benders Decomposition for SPIP

The upgrade state configuration $\boldsymbol{s}$ is a vector of length $|U|$ that comprises the operational state of every upgrade. For instance, an upgrade state configuration when four upgrades are non-operational, $\boldsymbol{s} = (0, 0, 0, 0)$, and when four upgrades are operational, $\boldsymbol{s} = (1, 1, 1, 1)$. Each upgrade state configuration $\boldsymbol{s}$ results in a unique network with different operational components. Thus an interdiction strategy must be solved for every $\boldsymbol{s} \in S$, where $S$ is the set of all upgrade state configurations.

| | |
|---|---|
| $\boldsymbol{s}$ | Vector indicating the operational status of all upgrades, $u \in U$ |
| $s_u \in \{0, 1\}$ | The state of upgrade $u$ where |

$$s_u = \begin{cases} 0 & \text{if } u \text{ non-operational (i.e., } p = 1) \\ 1 & \text{if } u \text{ operational (i.e., } p = 2) \end{cases}$$

| | |
|---|---|
| $\boldsymbol{s} \in S$ | Set of enumerated upgrade state configurations, where $\lvert S \rvert = 2^{\lvert U \rvert}$ |

To solve the optimal attack strategy for every upgrade state configuration, we utilize a basic decomposition algorithm. Algorithm 1 modifies the Benders decomposition algorithm presented by Israeli and Wood (2002) to solve the interdiction strategy for each upgrade state configuration $\boldsymbol{s} \in S$. The goal of Algorithm 1 is to quickly solve the SPIP by sequentially solving smaller instances instead of one very large instance. This is accomplished by decomposing the SPIP into two separate problems, the sub-problem (SP) and the master problem (MP). The SP handles the shortest path calculation and the MP handles the interdiction calculation. The algorithm iteratively solves the SP in line 8 to obtain a lower bound $LB$. Then, in line 14, it solves the MP to find an upper bound $UB$. Iterations continues until $LB$ and

$UB$ converge to a user-specified optimality tolerance, $\epsilon \geq 0$.

Algorithm 1 begins with the input of upgrade state configuration $\boldsymbol{s}$. Line 1 defines the graph $G(\boldsymbol{s})$ containing only operational arcs $A(\boldsymbol{s})$ and operational nodes $N(\boldsymbol{s})$ based on upgrade state configuration $\boldsymbol{s}$. Next, variables are initialized in steps 2 – 6. Most notable are variables $\hat{Y}$ and $\hat{x}$. Set $\hat{Y}$ stores a set of simple s-t paths that form a sub-graph of $G(\boldsymbol{s})$, and $\hat{x}$ stores an interdiction strategy. A simple s-t path is a path between the source node and terminus node that does not contain repeating vertices. The decomposition algorithm solves the SPIP based on smaller sub-graphs generated by $\hat{Y}$ rather than the entire graph $G(\boldsymbol{s})$, thereby decreasing computation time. The variables $LB$ and $UB$ store the objective values returned by $\text{SP}(G(\boldsymbol{s}), \hat{x})$ and $\text{MP}(\hat{Y})$, respectively.

**Algorithm 1** Benders Decomposition Algorithm for SPIP

**Input:** $\boldsymbol{s}$ ▷ Upgrade state configuration
**Output:** $z^*$, $\hat{x}$ ▷ Optimal objective value and interdiction strategy to the SPIP

1: $G(\boldsymbol{s})$ ▷ Graph containing only operational arcs given $\boldsymbol{s}$
2: $\epsilon \leftarrow 10^{-4}$
3: $UB \leftarrow \infty$
4: $LB \leftarrow 0$
5: $\hat{Y} \leftarrow \emptyset$
6: $\hat{x} \leftarrow \emptyset$
7: **while** $UB - LB > \epsilon$ **do**
8: $(\hat{y}, z_{\hat{x}}) \leftarrow$ `Solve` $(\text{SP}(G(\boldsymbol{s}), \hat{x}))$
9: $\hat{Y} \leftarrow \hat{Y} \cup \hat{y}$
10: $LB \leftarrow z_{\hat{x}}$
11: **if** $UB - LB \leq \epsilon$ **then**
12: Break while
13: **end if**
14: $(\hat{x}, z_{\hat{Y}}) \leftarrow$ `Solve` $(\text{MP}(\hat{Y}))$
15: $UB \leftarrow z_{\hat{Y}}$
16: **end while**
17: Return objective value $z^* = \frac{(UB-LB)}{2}$ and interdiction strategy $\hat{x}^*$.

`Solve` $(P)$ calls a solver; returns an objective value, $z$, and solution, $x$, to problem $P$.

The sub-problem $\text{SP}(G(\boldsymbol{s}), \hat{x})$ is given by equations (4a) – (4c). This formulation seeks a shortest path of graph $G(\boldsymbol{s})$ under the interdiction strategy $\hat{x}$. The interdiction strategy specifies the arc delays $d_{ijb}$. Once $\text{SP}(G(\boldsymbol{s}), \hat{x})$ is solved, the set of shortest path arcs $\hat{y}$ and the shortest path length $z_{\hat{x}}$ are returned. The returned shortest path arcs $\hat{y}$ are added to the set of arcs that form all s-t paths $\hat{Y}$ in line 9, and $LB$ is updated with the shortest path length in line 10. The updated $\hat{Y}$ is then sent to $\text{MP}(\hat{Y})$ to calculate a new interdiction strategy.

$$\text{SP}(G(\boldsymbol{s}), \hat{x}): \quad z_{\hat{x}} = \min_{y} \sum_{(i,j)\in A(\boldsymbol{s})} \sum_{b\in B} (c_{ij} + d_{ijb} \cdot \hat{x}_{ijb})\ y_{ij} \tag{4a}$$

$$\text{s.t.} \sum_{j:(i,j)\in A(\boldsymbol{s})} y_{ij} \ - \sum_{j:(j,i)\in A(\boldsymbol{s})} y_{ji} \geq f_i,\ \forall\ i \in N(\boldsymbol{s}), \tag{4b}$$

$$y_{ij} \geq 0,\ \forall\ (i,j) \in A(\boldsymbol{s}). \tag{4c}$$

$\text{MP}(\hat{Y})$ solves the interdiction strategy on the collection of s-t paths $\hat{Y}$. We utilize a set of constraints known as Benders cuts to maximize the shortest path. The constraints that form the set of Benders cuts is found in equation (5b). In equation (5b), a new constraint is added for each s-t path $\hat{y}$ in the collection of simple s-t paths $\hat{Y}$, where $z$ must be less than or equal to every post-interdiction path length. Therefore, by maximizing $z$, the lower-bound of interdicted path lengths is increased to the fullest extent. Subsequently, the shortest interdicted path length is maximized for all s-t paths in $\hat{Y}$. The budget constraint (5c) ensures the interdiction strategy is within the attacker's budget $b^{\text{atk}}$. The constraints only sums over arcs subject to interdiction; these arcs include those composing the set of s-t paths in $\hat{Y}$. We denote this subset of arcs as $A(\hat{Y})$. Once the updated interdiction strategy $\hat{x}$ is returned to Algorithm 1, the $UB$ is updated with the post-interdiction shortest path length. If $UB - LB < \epsilon$, then Algorithm 1 terminates with an optimal interdiction strategy to

SPIP($\boldsymbol{s}$). Otherwise, the iteration continues by seeking a new shortest path with the updated interdiction strategy $\hat{x}$.

$$\text{MP}(\hat{Y}): \qquad z_{\hat{Y}} = \max_{x} z \tag{5a}$$

$$\text{s.t.} \quad z \leq \sum_{(i,j)\in\hat{y}} \sum_{b\in B} (c_{ij} + d_{ijb} \cdot x_{ijb}), \ \forall\ \hat{y} \in \hat{Y}, \tag{5b}$$

$$\sum_{(i,j)\in A(\hat{Y})} \sum_{b\in B} b \cdot x_{ijb} \leq b^{\text{atk}}, \tag{5c}$$

$$\sum_{b\in B} x_{ijb} \leq 1, \ \forall\ (i,j) \in A(\hat{Y}), \tag{5d}$$

$$x_{ijb} \in \{0,1\}, \ \forall\ (i,j) \in A(\hat{Y}), \ \forall\ b \in B. \tag{5e}$$

### 3.3.3 Step 2: Modified Scheduling Formulation

The modified scheduling problem seeks an optimal ordering of upgrade state configurations, $\boldsymbol{s} \in S$ subject to constraints (1b) – (1g). In step 1, we obtained the optimal interdiction objective values $z^*$ for every $\boldsymbol{s} \in S$. These objective values are then stored in parameter $v_{\boldsymbol{s}}$ and passed to the modified scheduling problem. Furthermore, an additional decision variable $q_{\boldsymbol{s}t}$ is included in the formulation. This variable stores the decision to implement upgrade state configuration $\boldsymbol{s}$ during time period $t$.

| | |
|---|---|
| $v_{\boldsymbol{s}}$ | Optimal interdiction objective value for every $\boldsymbol{s} \in S$ |
| $q_{\boldsymbol{s}t}$ | Binary; 1 if $\boldsymbol{s}$ is implemented at time $t$, 0 otherwise |

In order to link the new decision variable $q_{\boldsymbol{s}t}$ with the existing decision variable $w_{upt}$, we include constraint (6b) in the modified formulation. Constraint (6b) establishes upgrade $u$ as operational (i.e., $w_{upt} = 1$ when $p = 2$) if both $q_{\boldsymbol{s}t}$ and $s_u$ equal 1. When $q_{\boldsymbol{s}t} = 1$, this indicates that upgrade state configuration $\boldsymbol{s}$ is implemented. When $s_u = 1$, this indicates that the individual upgrade within upgrade state con-

figuration $\boldsymbol{s}$ is operational. The remaining constraints in the upper-level scheduling formulation (1b) – (1g) are included in this formulation without alteration.

$$\min_{q} \sum_{\boldsymbol{s} \in S} \sum_{t \in T} r_t \ (v_{\boldsymbol{s}} \cdot q_{\boldsymbol{s}t}) \tag{6a}$$

$$\text{s.t. } s_u \cdot q_{\boldsymbol{s}t} \leq w_{u2t}, \ \forall \ \boldsymbol{s} \in S, \ \forall \ u \in U, \ \forall \ t \in T, \tag{6b}$$

$$(1b) - (1g). \tag{6c}$$

The original objective function (1a) is substituted with objective function (6a). Objective function (6a) reduces the impact of network interdiction by minimizing the risk $r_t$ of interdictions with objective values $v_{\boldsymbol{s}}$ for the corresponding upgrade state configurations $q_{\boldsymbol{s}t}$.

## 3.4 Heuristic Solution

### 3.4.1 Overview

Many large non-deterministic polynomial-time (NP) problems cannot be solved within an acceptable amount of time using existing optimization techniques. When problem instances become too large to optimally solve, heuristics are used to speed up computation time. A heuristic refines solutions to combinatorial problem by exploring the solution space via a series of moves to identify new solutions. A move involves a small change to the current solution to create a different solution. When a move occurs, we call the new solution a neighbor. A neighbor is defined as a solution that is a single move away from the current solution. Likewise, a neighborhood is defined as the set of all neighbors to the current solution. A heuristic discovers high quality solutions by performing moves that intensify on local optima and diversify to escape local optima. Heuristics return near-optimal solutions but cannot guarantee

optimality. This trade-off between time savings and a sub-optimal solution is generally accepted in real-world applications.

One such heuristic is Kirkpatrick et al.'s (1983) simulated annealing (SA) algorithm. The usefulness of the SA algorithm was demonstrated when Kirkpatrick et al. used it to solve the NP-hard traveling salesman problem (TSP). When Kirkpatrick et al. (1983) was published, the largest optimally solved TSP problem contained only 318 nodes. Kirkpatrick et al. (1983) apply the SA algorithm to a TSP size of 6,000 nodes to reach an approximate solution for this much larger problem size. This contribution was revolutionary for the field of heuristics.

The inspiration of the SA algorithm is derived from the physical annealing of metals (Kirkpatrick et al., 1983). The physical annealing process requires the initial heating of metal to a very high temperature followed by a slow cooling. This procedure alters the physical and chemical properties to increase the metal's ductility making the metal more workable. The SA algorithm uses "temperature" to control the probability of escaping local optima, thus exploring new neighborhoods. By initializing the algorithm with a high temperature and then slowly cooling it, SA converges on a high quality solution.

### 3.4.2 SA Algorithm for NUSP

To contextualize SA within the NUSP, we define a solution as an upgrade schedule identified by a prioritized sequence of upgrades. As we change the priority of an upgrade, the upgrade schedule changes. For instance, if the highest priority upgrade changes to the lowest priority upgrade, the acquisition start time of that upgrade changes from the first window of opportunity to the last window of opportunity. Therefore, changes to the prioritized sequence of upgrades directly affect the upgrade schedule, thus creating different solutions. We define a move as a swapping of priority

between two upgrades within the sequence. By executing a move, we obtain a new solution. Two solutions that are distinguished from each other by a single swap operation are called neighbors. It follows that the neighborhood of a given solution consists of all solutions that occur by swapping the priority of any two upgrades in the sequence.

To construct a solution and calculate its objective value, we present Algorithm 3 in Appendix A which takes a prioritized sequence of upgrades and returns a condensed upgrade schedule. The algorithm schedules upgrades in accordance with the given upgrade priority. While maintaining the upgrade priority, Algorithm 3 condenses the timeline by scheduling upgrades for the first available time slot that also satisfies all budgetary and programmatic constraints. If there is not enough time to complete acquisition for an upgrade (this happens frequently for lower-priority upgrades), it and all subsequent upgrades are excluded from the schedule. In this way, Algorithm 3 is assured to return a feasible solution that minimizes the maximum interdiction delay, subject to the given upgrade priority. Finally, we determine the upgrade state configuration $\boldsymbol{s}$ for each time period $t \in T$ in the solution (i.e., upgrade schedule). Then, to calculate the solution's objective value, we solve Algorithm 1 for each $\boldsymbol{s}$ and sum the returned maximized shortest path lengths.

We utilize a SA algorithm, Algorithm 2, to solve the upper-level scheduling problem. Algorithm 2 begins by initializing variables such as temperature $T$, minimum temperature $t$, number of inner loop iterations $N$, and cooling rate $\lambda$. Also, we define the variable $M$ that determines the maximum number of iterations the solution can remain unchanged before terminating the algorithm. Once these variables are initialized, a random initial starting solution $s$ is determined. There are four variables to track solutions throughout the algorithm: $s_{current}$, $s_{prev}$, $s_{new}$, and $s^*$. The current solution is stored in $s_{current}$, the previous best solution is stored in $s_{prev}$, the new,

neighboring solution is stored in $s_{new}$, and the best solution is stored in $s^*$.

The SA algorithm performs swap moves to explore the solution space by visiting neighboring solutions. To determine the solution quality of the neighboring solution, we calculate the difference in objective values $\Delta E$ between the current solution and the neighboring solution. If the move to a neighboring solution results in an improving objective value (i.e., $\Delta E < 0$), the move is accepted with 100% probability. However, if the move to a neighboring solution results in an equal or worsening solution (i.e., $\Delta E \geq 0$), the probability of accepting the move occurs with a probability $e^{-\Delta E/T}$. The probability of accepting a worse solution is dependent on both $T$ and $\Delta E$. As $T$ decreases, the probability of accepting a worse solution also decreases. Therefore, when $T$ is high in early iterations, worse solutions are more likely to be accepted, thereby allowing an escape from local optima. When $\Delta E$ is a large positive number, this indicates a very bad change to solution quality, and the probability of accepting this solution decreases. On the contrary, a small, positive $\Delta E$ indicates a marginal decrease in solution quality, and the probability of accepting this solution is higher.

There are two functions responsible for returning a new solution. The first function, `random_soln`() on line 7, randomly orders the prioritized upgrade sequence and returns a random initial solution. The second function, `swap`($s$) on line 12, performs a move by swapping the priority of two upgrades within solution $s$ and returns a neighboring solution. The algorithm terminates when $T$ reaches the minimum temperature $t$, or if the number of iterations where the solution remains unchanged $m$ reaches the maximum value $M$. Upon termination, the best solution $s^*$ is returned.

**Algorithm 2** Simulated Annealing Algorithm

```
1: T          ← 100                                  ▷ initial temperature
2: t          ← 1                                    ▷ minimum temperature
3: N          ← 40                                   ▷ number of inner-loop iterations
4: λ          ← 0.8                                  ▷ cooling rate
5: M          ← 10                                   ▷ maximum unchanged iterations
6: m          ← 0                                    ▷ unchanged iterations counter
7: s_current ← random_soln()
8: s*         ← s_current
9: while t < T and m < M do
10:     s_prev ← s*
11:     for n = 1 to N do
12:         s_new ← swap(s_current)
13:         ΔE ← z(s_new) − z(s_current)
14:         if ΔE < 0 then
15:             s_current ← s_new
16:             if z(s_current) < z(s*) then
17:                 s* ← s_current
18:             end if
19:         else
20:             ρ ← random_num(0, 1)
21:             if ρ ≤ e^(−ΔE/T) then
22:                 s_current ← s_new
23:             end if
24:         end if
25:     end for
26:     if z(s_prev) ≡ z(s*) then
27:         m ← m + 1
28:     else
29:         m ← 0
30:     end if
31:     T ← λ · T
32: end while
33: Return s*
```

`random_soln`() is a call to a function that randomizes the upgrade sequence to return a random solution $s$ and an objective value $z(s)$.
`swap`($s$) is a call to a function that performs a swap move on solution $s$ and returns a neighboring solution $s_{new}$ and an objective value $z(s_{new})$.
`random_num`(x, y) is a call to a random number generator that returns a random value between x and y, exclusively.

## 3.5 Other Considerations

### 3.5.1 Node Interdiction

The formulation only supports arc interdiction; however, a node interdiction is imitated through the creation of a duplicate node. All arcs that exit the original node are reassigned to exit the new, duplicate node. The original and duplicate nodes are joined by a zero length arc ($c_{ij} = 0$). An interdiction to the original node occurs when the adjoining arc is attacked. For instance, Figure 1 represents a network prior to the transformation. Figure 2 represents the transformed network that allows a node interdiction. In this example, the interdiction of arc (3, 3') imitates an interdiction to node 3.

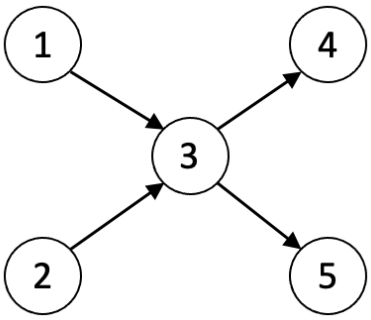


**Figure 1. Network Before Node Interdiction Transformation**

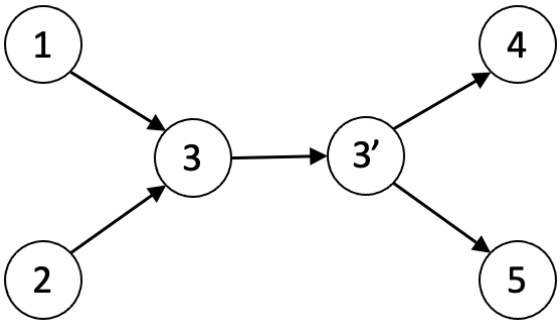


**Figure 2. Network After Node Interdiction Transformation**

### 3.5.2 Multiple Acquisition Phases

It may be desirable to model more than two phases. For instance, the Department of Defense (DoD) utilizes a multi-phase acquisition process composed of Research and Development (R&D), Production and Deployment (P&D), and Operations and Support (O&S) phases. Since each phase has its own appropriation category; separate budget constraints are required for each phase. Additionally, the lead time required to complete each phase is also required. Therefore, we modify the lead time parameter from $\ell_u$ to $\ell_{up}$.

| | |
|---|---|
| $p \in P$ | Set of acquisition phases and operations phase, $\{\text{R\&D acquisition} = 1, \text{P\&D acquisition} = 2, \text{O\&S operations} = 3\}$ |
| $\ell_{up}$ | Lead time for upgrade $u$ to complete acquisition for each acquisition phase $p \in P \setminus \{3, \text{O\&S operations}\}$ |

Lastly, two constraints must be altered due to the modification of parameter $\ell$. The original constraints (1d) and (1e) are changed to the following:

$$\ell_{up} \cdot w_{up+1t} \leq \sum_{t'=1}^{t} w_{upt'}, \ \forall\ u \in U,\ p \in P \setminus \{3\},\ \forall\ t \in T, \tag{1d'}$$

$$\sum_{t \in T} w_{upt} \leq \ell_{up},\ \forall\ u \in U,\ p \in P \setminus \{3\}. \tag{1e'}$$

### 3.5.3 Arc Sever

Sufficiently large values M within delay matrix $d_{ijb}$ enable arc severs if these values are selected as part of the interdiction strategy. A value of M is guaranteed to be sufficiently large if it is bigger than the longest, simple s-t path in the network. A simple s-t path is a path between the source node and terminus node that does not contain repeating vertices.

# IV. Testing, Results, and Analysis

## 4.1 Background

This chapter presents a network instance, a solution quality study, and a heuristic tuning excursion. The solution quality study investigates the effects of the defender's risk preferences and attacker's budget on the upgrade schedule and the resilience of the network. To examine these effects, we introduce four scenarios that differ in attacker budget and risk preferences and analyze how their unique solutions compare to a myopic solution. Finally, a heuristic tuning excursion is presented to evaluate the performance of the SA algorithm with respect to run time and optimality gap.

The analysis presented in this chapter is performed on a MacBook Pro with a 2.3 GHz Quad-Core Intel i5 processor and 8 GB of random access memory (RAM). The algorithms were written in Python version 3.7 and used Gurobi version 9.1 as the solver and NetworkX version 2.5 as the network generator.

## 4.2 Network Instance

In both the solution quality study and heuristic tuning excursion, we use a transportation network to imitate a hypothetical NC3 modernization problem. The network instance is instantiated with data from the transportation network of Rome, Italy (Storchi et al., 1999). The data contains every $(i, j)$ node pair and the associated arc distance between node $i$ and node $j$ and consists of 3353 nodes and 8870 arcs. For the arc lengths $c_{ij}$ we use the distances provided in the data set.

### 4.2.1 Arc Delays

We randomly generated values for the attacker's interdiction delays, denoted as $d_{ijb}$, for every arc $(i, j)$. Table 2 shows the $d_{ijb}$ values for a subset of the network arcs.

We generate the full matrix by populating it with random integers drawn uniformly from the range [0, 15]. Each row is sorted in non-decreasing order to appropriately account for an increase in delay as the attacker spends additional resources.

**Table 2. Sample of Attacker's Delay Matrix**

| | Resources ($b$) | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arc $(i, j)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| (2958, 2959) | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 6 | 6 | 6 | 7 | 8 | 8 | 8 | 8 |
| (2958, 2953) | 0 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 5 | 6 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 | 9 | 9 |
| (2958, 2963) | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 3 | 6 | 6 | 7 | 7 | 7 | 8 | 8 | 8 | 9 |
| (2959, 2960) | 0 | 0 | 0 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 7 | 7 | 7 | 8 | 8 | 9 |
| (2959, 2958) | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 8 | 8 | 9 | 9 | 9 | 10 | 11 | 12 | 12 |
| (1494, 1573) | 0 | 1 | 2 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 8 | 9 |
| (1494, 1481) | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 5 | 5 | 6 | 7 | 7 | 8 | 8 | 8 | 9 | 9 |
| (1494, 1493) | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 7 | 7 | 8 | 8 | 9 | 9 |
| (1573, 1497) | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 4 | 5 | 5 | 5 | 8 | 8 | 9 | 9 | 10 | 11 | 11 |
| (1573, 1494) | 0 | 0 | 0 | 0 | 2 | 2 | 4 | 4 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 7 | 7 | 7 | 8 | 9 |

### 4.2.2 Upgrades and Budget

We built our upgrades from the existing network by identifying several connected components along the shortest paths. To create these connected components, first we identify the highest degree node in the shortest path between the source and terminus nodes. Then, we add every arc connected to the highest degree node to the set of non-operational arcs. The shortest path is recalculated after removing the set of non-operational arcs from the network and the process repeats. Once a sufficient number of non-operational arcs are generated, then we identify the non-operational connected components. The non-operational connected components are identified using the *weakly_connected_components* NetworkX function (Hagberg et al., 2008). This function is passed a graph formed by the set of non-operational arcs. The function returns the set of non-operational connected components. Each non-operational connected component is uniformly distributed among 10 upgrades.

Upgrades to the NC3 network go through the DoD acquisition process as detailed

in Section 3.5.2. Each phase in the acquisition process has its own budget because of the different appropriation categories in the DoD. Therefore, research and development (R&D), production and deployment (P&D), and operations and sustainment (O&S) are separate budgets. This means that separate budgetary constraints for each phase must be satisfied to reach a feasible upgrade schedule.

Table 3 shows the annual cost of each phase for every upgrade. Also, each upgrade is assigned a lead time to complete the R&D phase and a lead time to complete the P&D phase. Furthermore, Table 3 outlines the annual budget for each phase. Due to budget constraints, not every upgrade can start acquisition at the same time. In fact, only a single upgrade may enter the R&D phase or the P&D phase at a time. This highly constrained budget is intentional. In doing so, the upgrade schedule solution provides a clear prioritization of upgrades and simplifies analysis. The operations budget is another limiting factor that prevents every upgrade from being scheduled since the O&S budget is less than the sum of O&S cost.

**Table 3. Upgrade Cost, Lead Time, and Defender's Budget**

| Upgrade | # Arcs | R&D Cost | P&D Cost | O&S Cost | R&D Yrs | P&D Yrs |
|---|---|---|---|---|---|---|
| 1 | 132 | 200 | 250 | 214 | 1 | 3 |
| 2 | 110 | 171 | 212 | 173 | 1 | 4 |
| 3 | 94 | 150 | 185 | 207 | 3 | 2 |
| 4 | 96 | 152 | 188 | 190 | 1 | 1 |
| 5 | 84 | 136 | 167 | 197 | 1 | 2 |
| 6 | 72 | 121 | 147 | 215 | 3 | 3 |
| 7 | 68 | 115 | 140 | 163 | 2 | 2 |
| 8 | 64 | 110 | 133 | 209 | 2 | 3 |
| 9 | 57 | 101 | 121 | 170 | 3 | 3 |
| 10 | 56 | 100 | 120 | 190 | 2 | 1 |
| | | R&D | P&D | O&S | | |
| Annual Budget | | 200 | 250 | 1500 | | |

To calculate the annual cost of R&D and P&D for each upgrade, first we initialize a minimum and maximum cost value. For R&D, the minimum and maximum cost

values are 100 and 200; the P&D minimum and maximum cost values are 120 and 250. The cost values for each upgrade are normalized within these ranges based on the number of arcs in the upgrade. In effect, the larger upgrades cost more than smaller upgrades. The O&S cost, however, is based on a random integer uniformly drawn from the range [150, 220].

Figure 3 highlights the set of arcs that belong to an upgrade. Figure 3 also labels the source node (bottom-left of Figure 3) and terminus node (top-right of Figure 3).

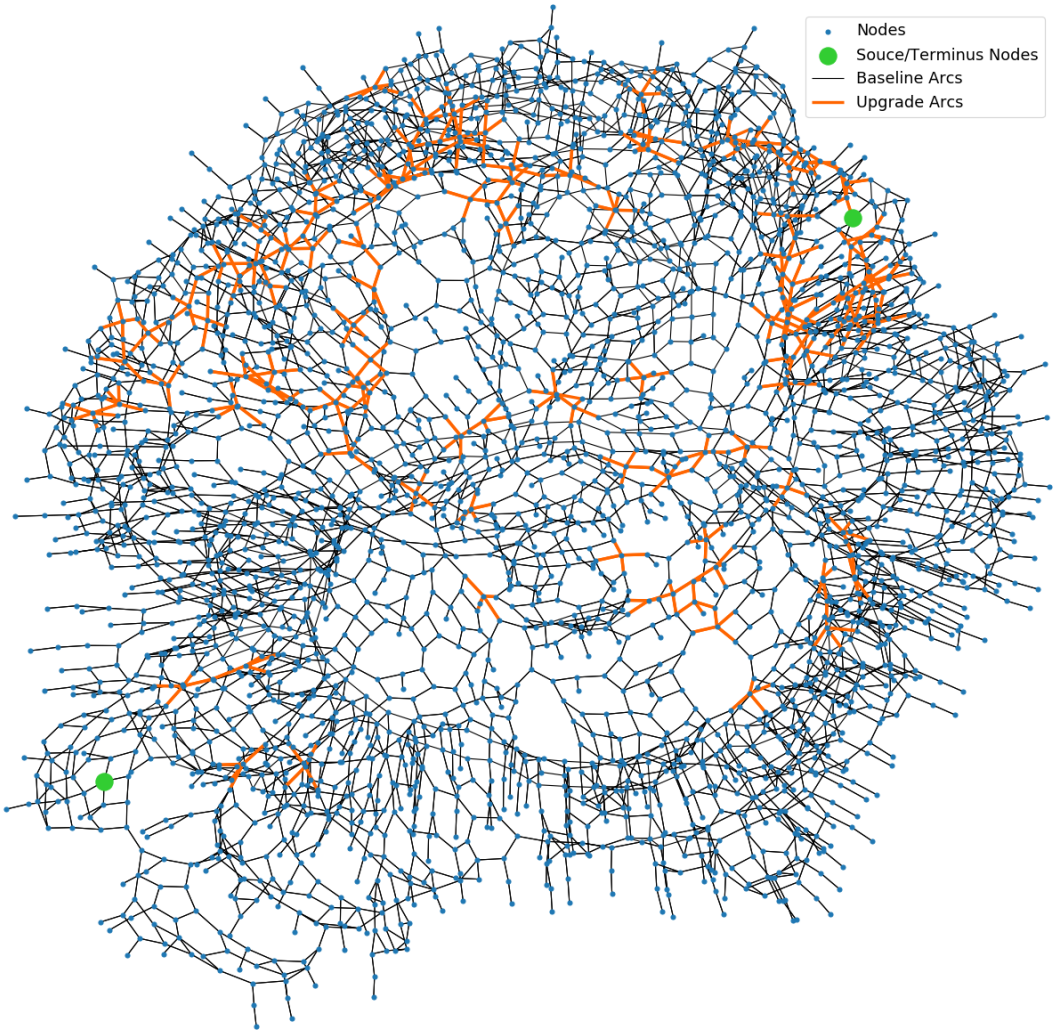


**Figure 3. Network Instance and Upgrade Arcs**

## 4.3 Solution Quality Study

### 4.3.1 Motivation

The solution quality study investigates how an optimal upgrade schedule is affected by the attacker's budget and the defender's risk preferences. By changing the attacker's budget, we compare a less capable adversary with a more capable adversary. The effects of this factor are motivated by Brown et al. (2005, 2006) and Alderson et al. (2013). These authors conclude that an optimal attacker's strategy is often unique to their budget, and a rule-of-thumb to determine the k-most vital arcs is insufficient. Based on their assessment, we expect the unique attack strategies from a wealthy attacker to result in a different upgrade schedule than the unique attack strategies from a less wealthy attacker. To put this into a real-world context, this is similar to comparing a potential adversary with more resources, like Russia, and one with fewer resources, like North Korea. Due to differences in resources, their interdiction strategies are likely different.

Our study also investigates the effects of the defender's risk preferences on an optimal upgrade schedule. The goal is to determine how short-term risk-averse schedules compare to long-term risk-averse schedules. If risk of attack in the short-term is the greatest concern, upgrades with shorter lead times are expected to be scheduled first. That is, an upgrade schedule is expected to adapt to different risk preferences and provide more resilience during times of higher risk.

### 4.3.2 Design

We introduce four scenarios to examine how risk preferences and the attacker's budget affect the optimal upgrade schedule, and consequently, the resilience of the network (i.e., post-interdiction shortest path length). Scenario 1 includes an attacker budget of 20 and places a higher risk of an attack in the second half of the upgrade

schedule. Scenario 2 deviates from Scenario 1 by reversing the risk values such that a higher risk of attack is assigned to the first half of the upgrade schedule. Scenario 2 maintains the same attacker budget as Scenario 1. The attacker's budget for scenarios 3 and 4 are set to 40. The risk preferences for scenarios 3 and 4 match those of scenarios 1 and 2, respectively. Table 4 displays the differences between scenarios. These scenario settings provide a full-factorial experimental design for our study.

**Table 4. Scenario Variable Comparison**

| Scenario | Risk Years 1-10 | Risk Years 11-20 | Attacker Budget |
|---|---|---|---|
| 1 | 1 | 10 | 20 |
| 2 | 10 | 1 | 20 |
| 3 | 1 | 10 | 40 |
| 4 | 10 | 1 | 40 |

To demonstrate the value of incorporating the interdictor's activities in our Stackelberg game, we create a myopic upgrade schedule as a benchmark for comparison. The myopic strategy does not consider how the attacker's interdiction strategy affects the shortest path. Recall that resilience is measured by the length of the shortest path *after* an interdiction occurs. The myopic strategy disregards this resilience measure and only considers the pre-interdiction shortest path improvement of each upgrade. The myopic upgrade schedule uses a greedy approach to prioritize upgrades. The upgrades that improve the pre-interdiction path length the most begin acquisition first. The myopic upgrade schedule, depicted in Figure 4, reveals how each upgrade is prioritized when implementing the myopic strategy. Figure 4 also indicates the time period each upgrade becomes operational. Upgrades 2, 5, 6, and 7 are not selected with the myopic strategy.

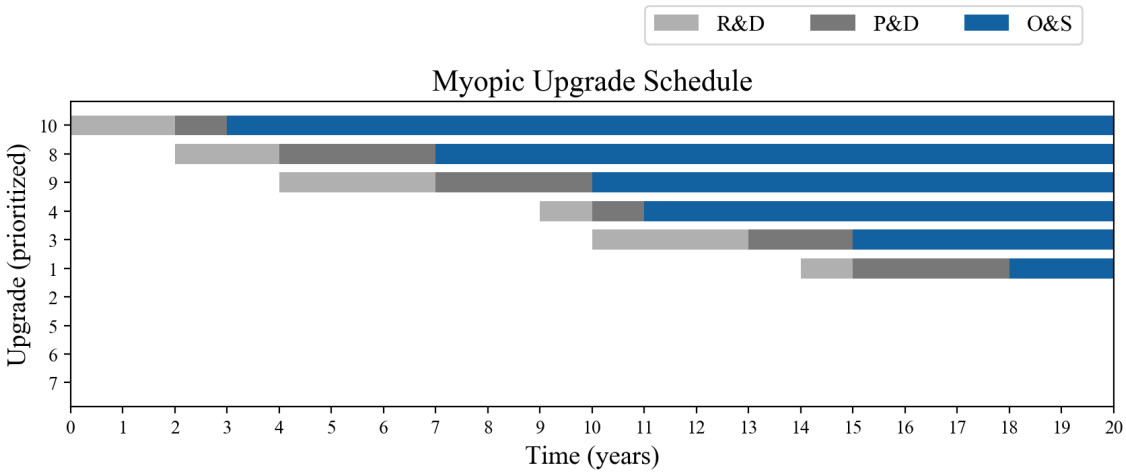


**Figure 4. Myopic Upgrade Schedule**

### 4.3.3 Evaluation Criteria

Two evaluation criteria are considered for comparing solutions. First, we include the upgrade schedule to convey the prioritized order of upgrades based on acquisition start time. Upgrade priority is important because it directly impacts how quickly network resilience improves during the upgrade schedule.

Second, we measure resilience of the network by calculating the post-interdiction shortest path length based on the set of operational upgrades at each time period. Note that a decrease to post-interdiction shortest path length indicates an increase in resilience. The post-interdiction shortest path length values are then plotted to create a resilience curve. The resilience curve is denoted as $z_\omega(t)$ for the set of optimal solutions obtained by solving scenarios $1-4$. We also calculate the "no change" curve $z_0(t)$ which outputs a constant interdicted path length when none of the upgrades are selected. Lastly, $z_M(t)$ gives the interdicted path lengths under the myopic upgrade schedule. Figures $5-8$ compare the resilience curve of each scenario $z_1(t)-z_4(t)$ to the "no change" curve $z_0(t)$ and the myopic curve $z_M(t)$. By measuring the area under these curves, we use the following equation to calculate the resilience improvement

percentage between the optimal and myopic upgrade schedules:

$$\text{Resilience Improvement } \% = \frac{\int_{t=0}^{20} z_0(t) - \int_{t=0}^{20} z_\omega(t)}{\int_{t=0}^{20} z_0(t) - \int_{t=0}^{20} z_M(t)} - 1$$

### 4.3.4 Results

The SPIP enumeration takes significantly more time to run than the upper-level scheduling problem. Table 5 reveals that attacker budget affects the time to solve the SPIP. In fact, by doubling the attacker budget, the time to solve the SPIP enumeration increases eightfold. The attacker budget does not contribute to the upper-level scheduling run time.

**Table 5. Scenario Run Time**

| | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 |
|---|---|---|---|---|
| SPIP Enumeration Time (s) | 1546.6 | 1546.6 | 12329.5 | 12329.5 |
| Upper-level Scheduling Time (s) | 85.4 | 34.1 | 87.6 | 99.3 |

Although each scenario produces a unique upgrade schedule, we glean valuable insights by studying the commonalities between the schedules. All scenarios prioritize upgrade 4 first and none of the scenarios select upgrades 3, 7, or 9. Also, upgrades 2 and 6 are always selected last, whereas the prioritization of upgrades 1, 5, 8, and 10 vary among each scenario. These observations suggest the following priorities:

| | |
|---|---|
| Priority 1 | 4 |
| Priority 2 – 5 | $\{1, 5, 8, 10\}$ |
| Priority 6 – 7 | $\{2, 6\}$ |
| Not Implemented | $\{3, 7, 9\}$ |

The variation between scenarios indicate that both the defender's risk preferences and attacker's budget affect the solution. Before providing analysis for risk preferences and attacker budget, this section compares the optimal solution of each scenario to that of the myopic.

The optimal upgrade schedule and resilience performance for Scenario 1 (20 attacker budget and long-term risk-averse schedule) are depicted in Figure 5. Results indicate a noticeable difference in upgrade priority and resilience performance. The upgrade priority for the myopic solution (10, 8, 9, 4, 3, 1, 2, 5, 6, 7) is vastly different than the optimal priority for Scenario 1's solution (4, 5, 8, 1, 10, 2, 6, 3, 7, 9). The only upgrade left out of both upgrade schedules is upgrade 7. The optimal solution for Scenario 1 achieves a resilience improvement of 8.2% over the myopic solution.

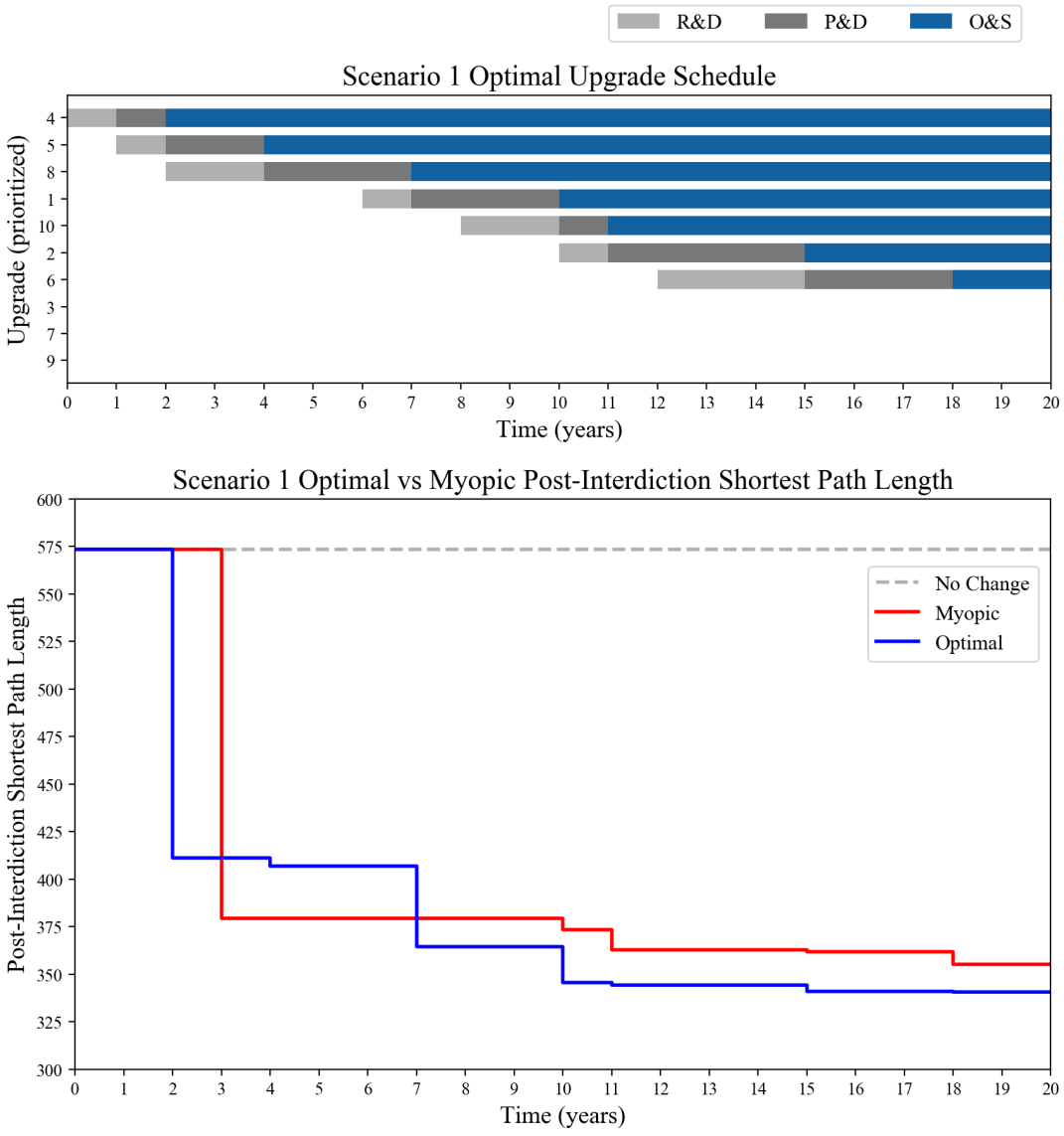


**Figure 5. Upgrade Schedule and Interdicted Path Length - Scenario 1**

The optimal upgrade schedule and resilience performance for Scenario 2 (20 attacker budget and short-term risk-averse schedule) are depicted in Figure 6. These graphics also reveal contrasting upgrade priorities. Scenario 2's post-interdicted shortest path length nearly dominates that of the myopic, except for year 3. The optimal solution for Scenario 2 achieves a resilience improvement of 11.9% over the myopic solution.

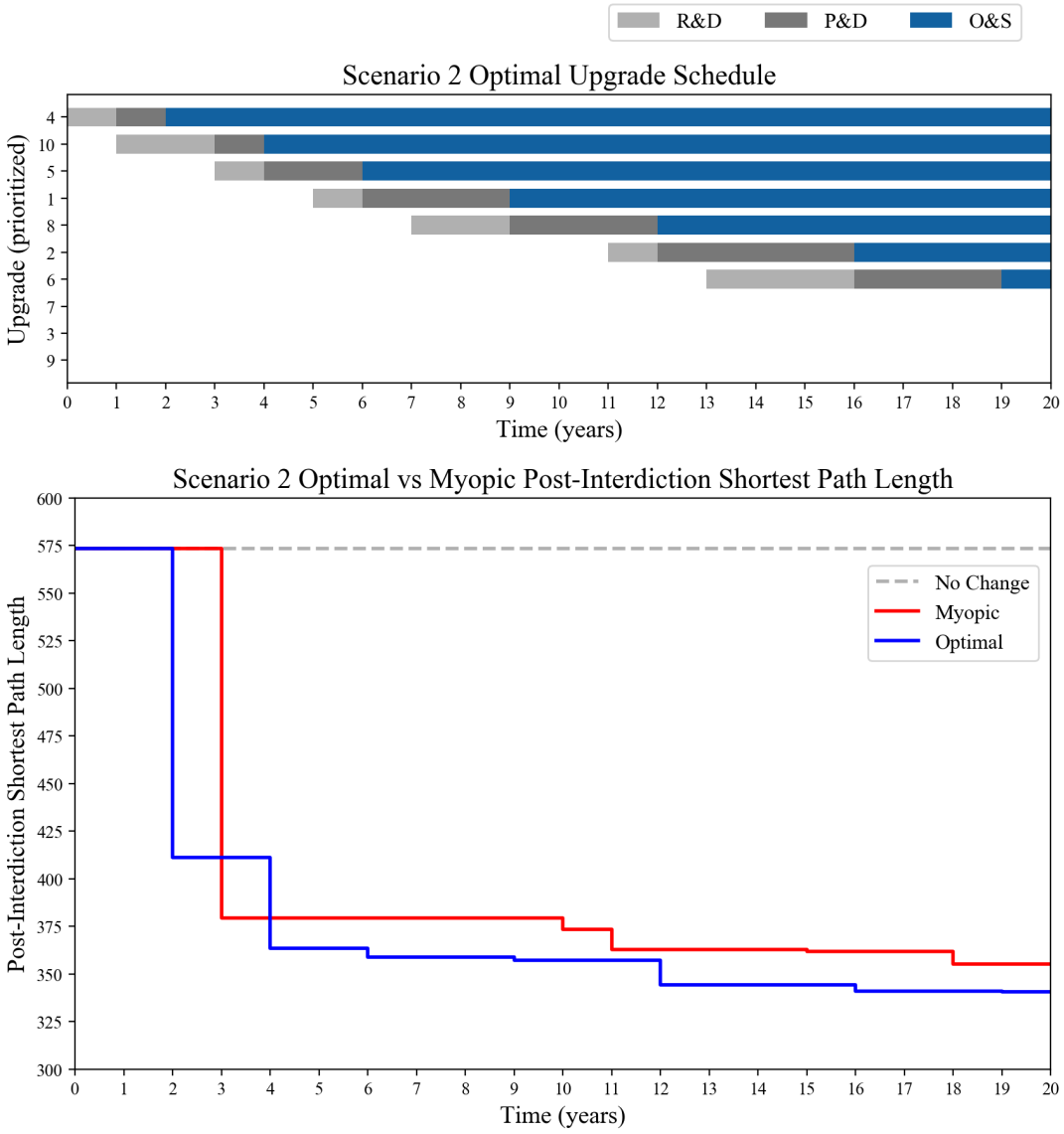


**Figure 6. Upgrade Schedule and Interdicted Path Length - Scenario 2**

The optimal upgrade schedule and resilience performance for Scenario 3 (40 attacker budget and long-term risk-averse schedule) are depicted in Figure 7. Results indicate a noticeable difference in upgrade priority and resilience performance. The optimal solution for Scenario 3 achieves a resilience improvement of 12.7% over the myopic solution.

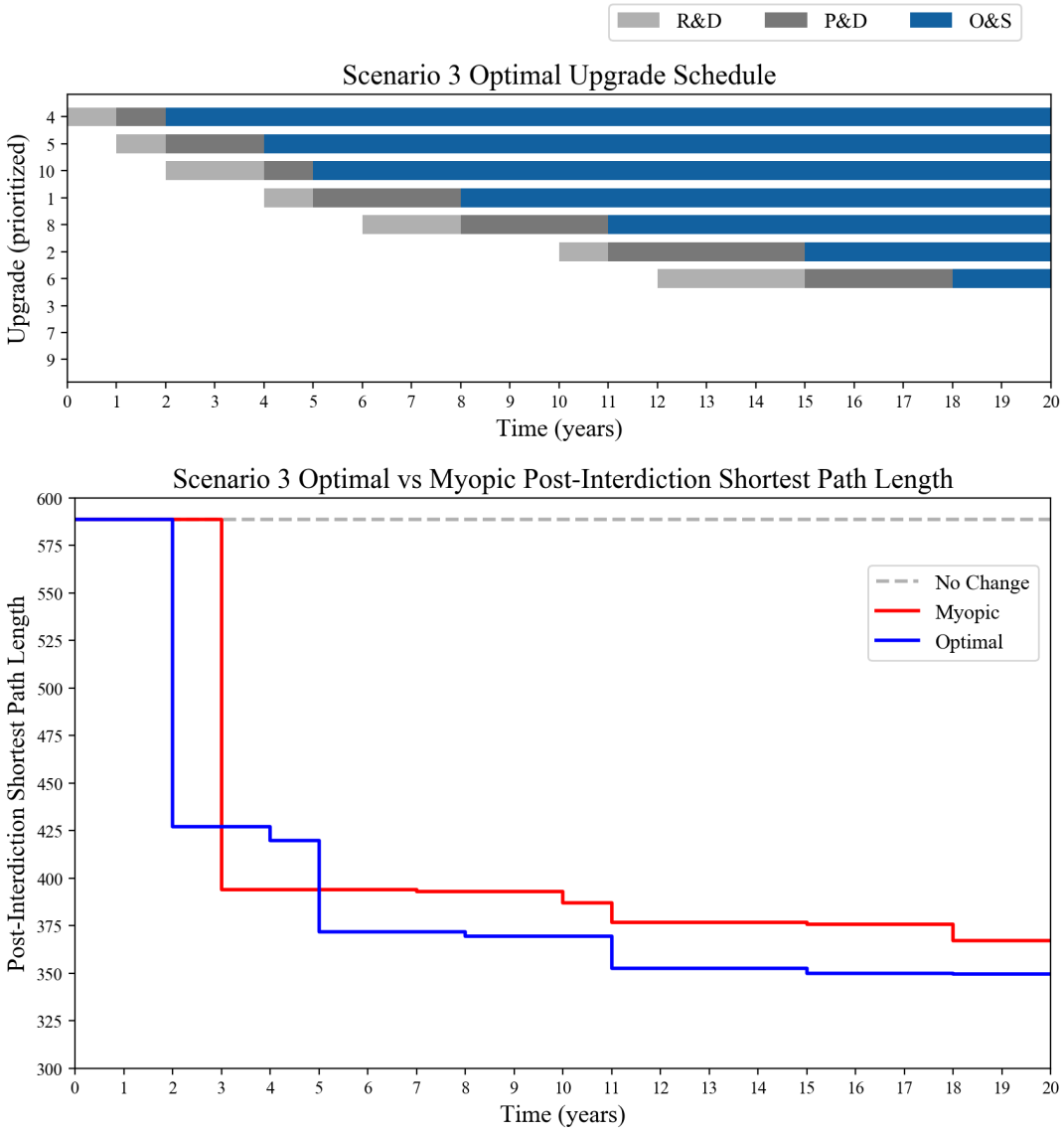


**Figure 7. Upgrade Schedule and Interdicted Path Length - Scenario 3**

The optimal upgrade schedule and resilience performance for Scenario 4 (40 attacker budget and short-term risk-averse schedule) are depicted in Figure 8. Results indicate a noticeable difference in upgrade priority and resilience performance. The optimal solution for Scenario 4 achieves a resilience improvement of 13.1% over the myopic solution.

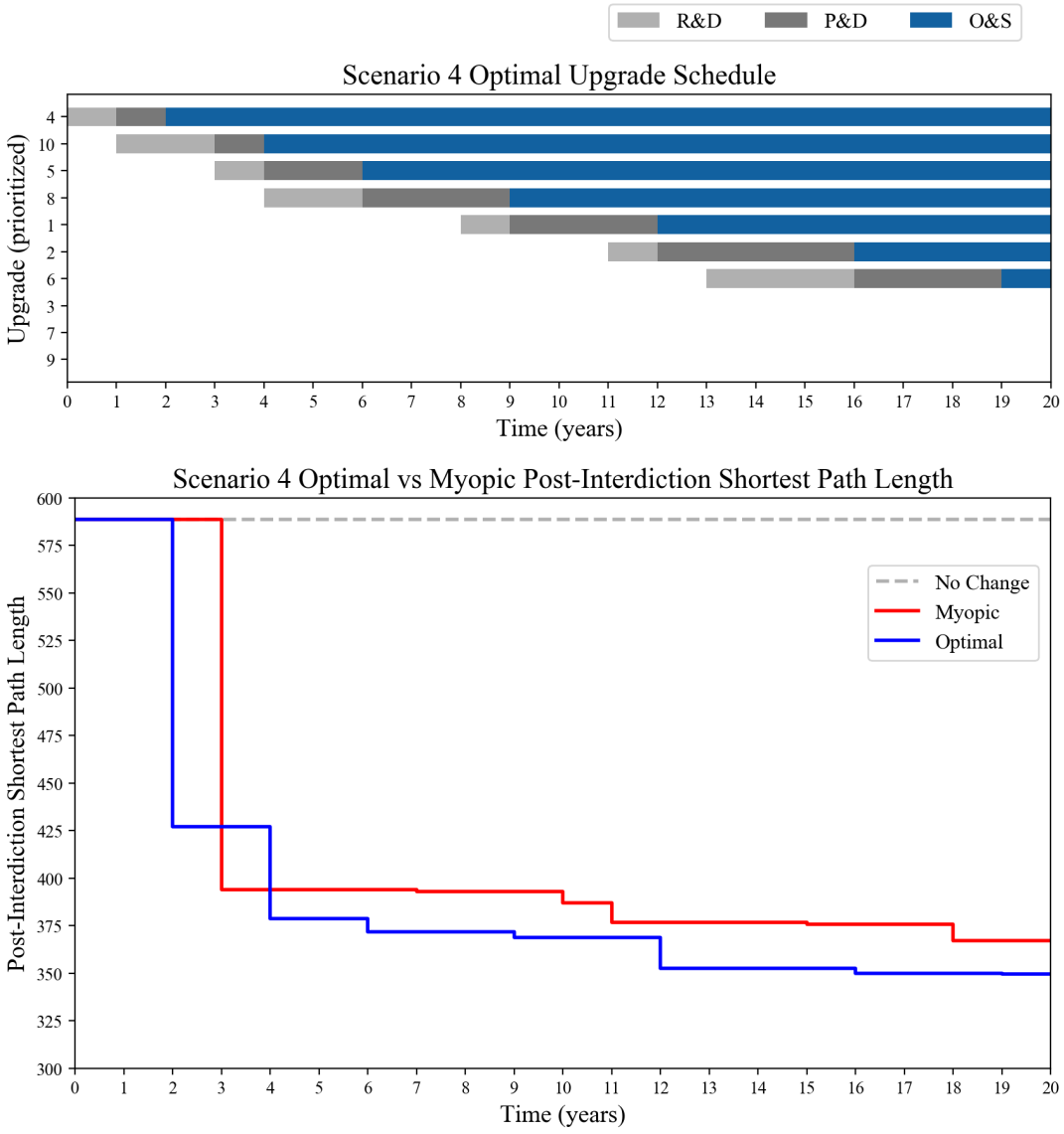


**Figure 8. Upgrade Schedule and Interdicted Path Length - Scenario 4**

Next, the solution quality study analyzes the effects of the defender's risk preferences on the optimal upgrade schedule. We determined that the short-term risk-averse solutions provide additional upfront resilience, whereas long-term risk-averse solutions provide more back-end resilience. This occurrence is observed by comparing short-term versus long-term risk preferences captured in Figure 9 and Figure 10.

Scenario 1 and 2 are compared in Figure 9 since these scenarios utilize different risk preferences but the same attacker budget. Scenario 1 implements a long-term risk-averse schedule, and Scenario 2 implements a short-term risk-averse schedule. The first 10 years of the graph show Scenario 2 outperforming Scenario 1 because of the reduced post-interdiction shortest path length. At year 10, Scenario 1 surpasses Scenario 2 and provides additional resilience in the second half of the schedule. Overall, Scenario 2 seems to provide more resilience than Scenario 1 due to the large gap between years 4 and 7; however, if an attack is 10 times more likely to occur in the second half of the schedule, then Scenario 1's upgrade schedule provides better resilience.

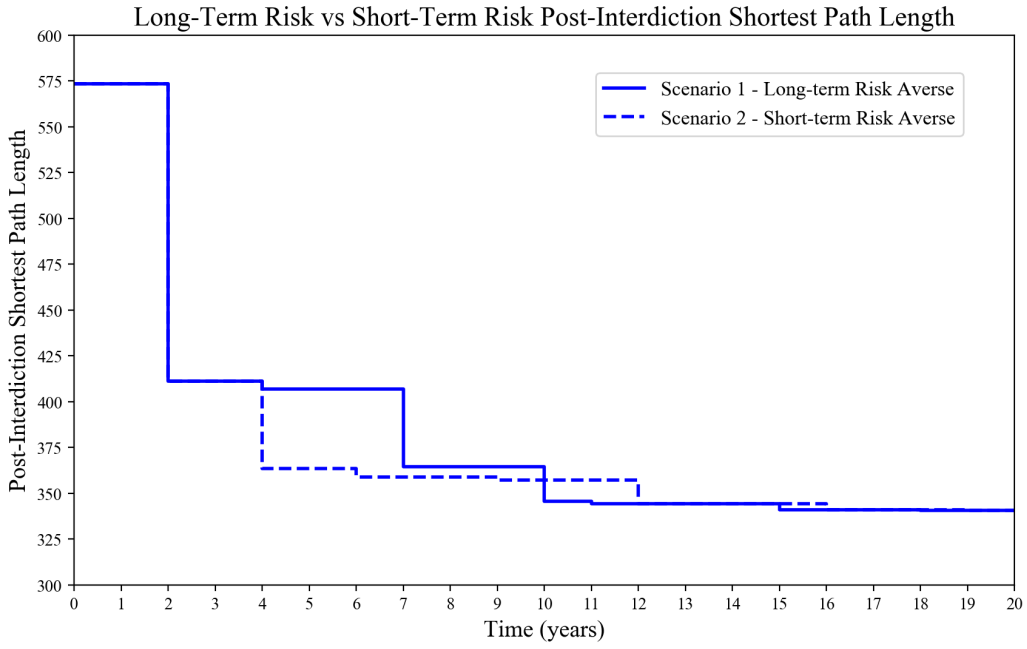


**Figure 9. Risk Comparison - Scenario 1 and 2 (Low Attacker Budget)**

The solutions to scenarios 3 and 4 respond in a similar manner. Scenario 3 provides a better long-term risk-averse schedule when compared to Scenario 4's short-term risk-averse schedule and vice versa. This is depicted in Figure 10. However, this difference is less obvious because Scenario 4 does not completely dominate Scenario 3 in the first half of the schedule.

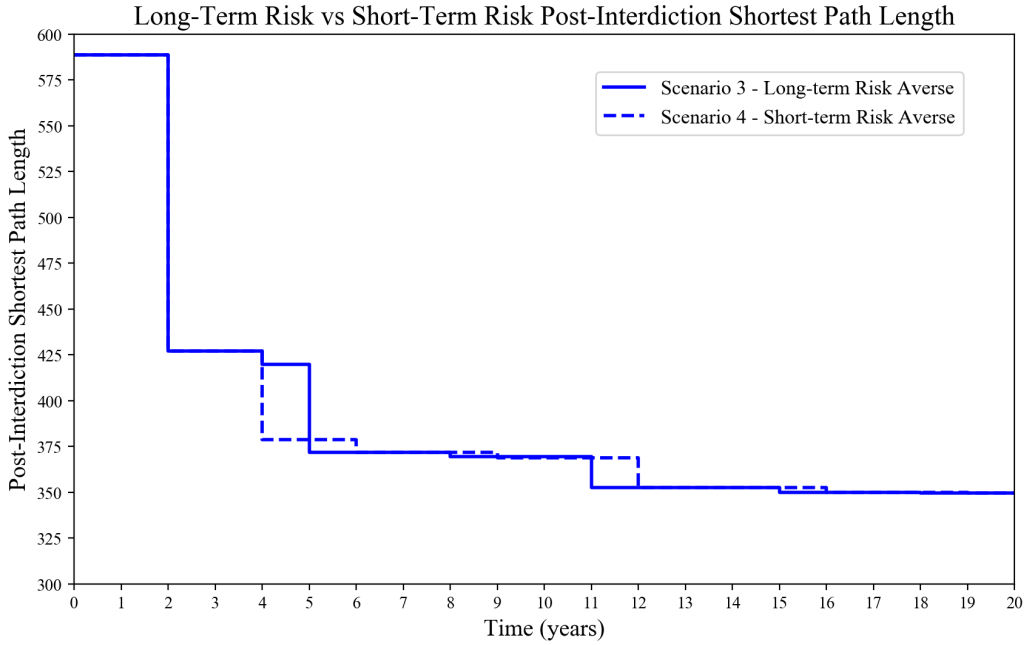


**Figure 10. Risk Comparison - Scenario 3 and 4 (High Attacker Budget)**

Next, we investigate the effects of the attacker's budget on the optimal upgrade schedule. It is important to align the upgrade schedule according to the attacker's budget. For instance, if the attacker's budget is 40 and the decision maker plans accordingly, then the Scenario 3 upgrade schedule is indeed the optimal solution. However, if the attacker's budget is 40 and the decision maker plans for a lower attacker budget, like that of Scenario 1, then the upgrade schedule is no longer the optimal solution. We demonstrate this by comparing the upgrade schedules of scenarios 1 and 3 since these maintain the same risk preferences but different attacker budgets. Figure 11 illustrates the interdicted path lengths when the decision maker

has correctly planned for an attacker budget of 40 (i.e., Scenario 3's upgrade schedule) or incorrectly planned for an attacker budget of 20 (i.e., Scenario 1's upgrade schedule). The plan that correctly identifies the attacker budget provides more resilience than the plan that incorrectly identifies the attacker's budget. This analysis suggests that a one-size-fits-all upgrade schedule may not be optimal for different attacker budgets.

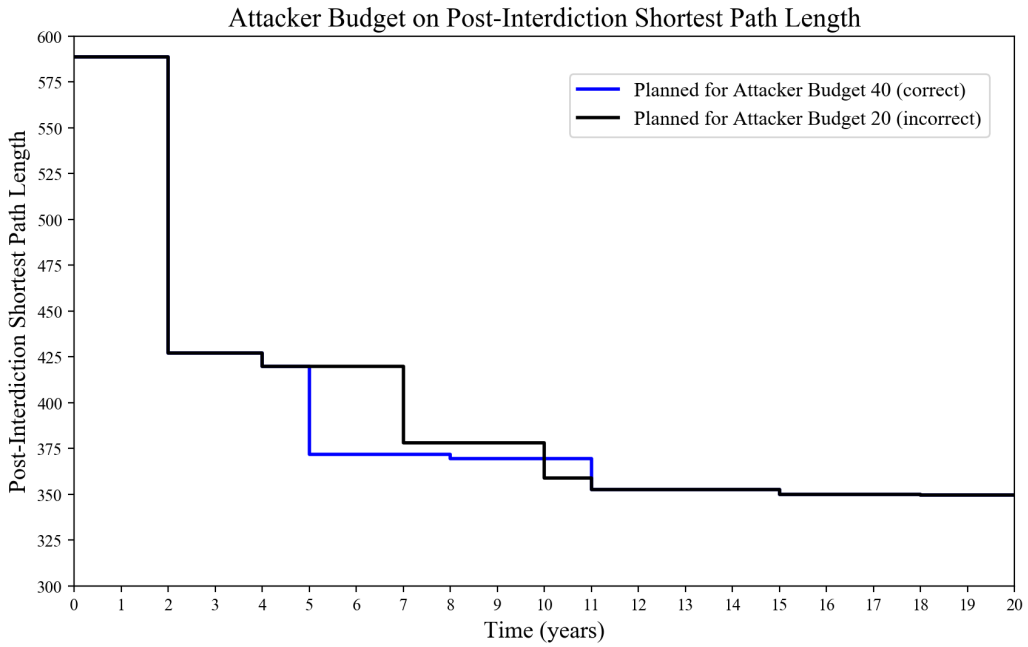


**Figure 11. Attacker Budget Planning Comparison**

To explain the dependence of the upgrade schedule on the attacker's budget, we present Figure 12 and Figure 13. Figure 12 illustrates the interdicted arcs from an attacker with a budget of 20. Figure 13 illustrates the interdicted arcs from the exact same attacker but with a budget of 40. The differences between interdiction strategies are circled between each figure. The blue circles highlight interdicted arcs that occur in Scenario 1 but not Scenario 3. The green circles highlight interdicted arcs that occur in Scenario 3 but not Scenario 1. These figures demonstrate why an incorrect assumption of the attacker's budget may result in a sub-optimal upgrade schedule.

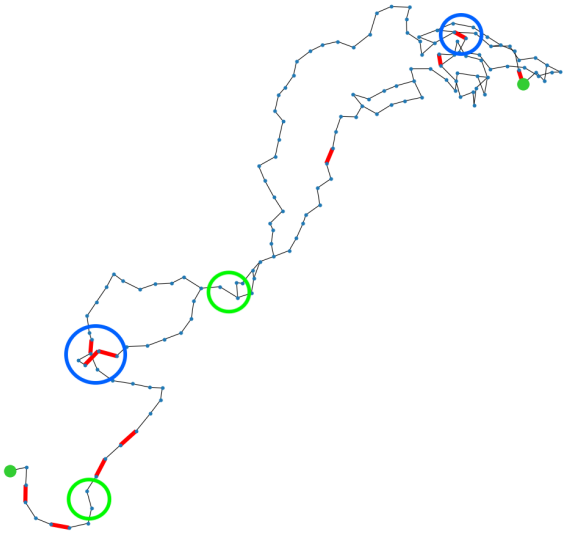

**Figure 12. Attack Scenario 1 (20 Attacker Budget)**

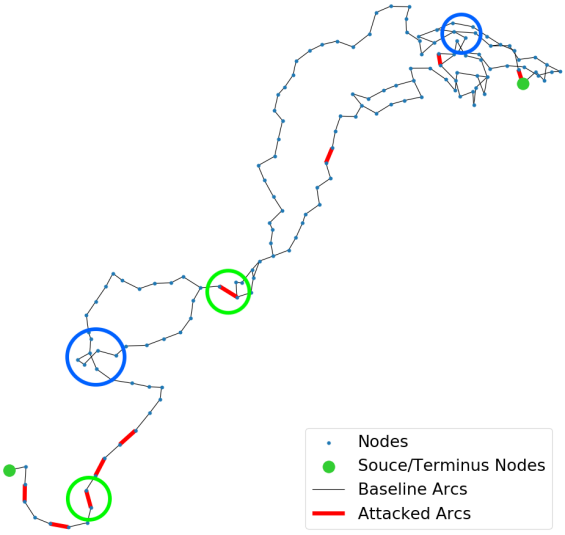


**Figure 13. Attack Scenario 3 (40 Attacker Budget)**

One would expect that the set of arcs interdicted with a budget of 20 is contained within the set of arcs interdicted with a budget of 40, but this is not the case. In fact, the optimal interdiction strategy when the attacker has a budget of 40 targets fewer arcs than the optimal attack when the budget is 20 (see Table 6). It is evident from the list of attacked arcs in Table 6 that neither interdiction strategy fully dominates the other. Therefore, it is invalid to assume that all interdictions strategies are realized when a large attacker budget is selected. The disparities between interdiction strategies cause the changes in optimal upgrade schedules. Thus, it is important to select an upgrade schedule tailored to an attacker's budget.

**Table 6. Comparison of Attacked Arcs with Different Attacker Budgets**

| Scenario | Attacker Budget | Same Attacked Arcs | Different Attacked Arcs |
|---|---|---|---|
| 1 | 20 | (1712, 1767), (1991, 1990), (271, 272), (2040, 2042), (126, 128), (221, 242), (133, 134) | (903, 985), (909, 993), (993, 994), (1993, 1986) |
| 3 | 40 | (1712, 1767), (1991, 1990), (271, 272), (2040, 2042), (126, 128), (221, 242), (133, 134) | (211, 220), (1137, 1161) |

### 4.4 Heuristic Tuning Excursion

The purpose of the heuristic tuning excursion is to identify a set of parameters that consistently return high-quality solutions from the SA algorithm. We conduct a computational experiment on Scenario 1 to determine the best performing set of parameters. The full factorial computational experiment consists of 108 different runs using the following values for each parameter:

| | | |
|---|---|---|
| $N$ | [10, 40, 80] | (Number of inner loops) |
| $T$ | [50, 100, 250, 500] | (Initial temperature) |
| $\lambda$ | [0.8, 0.9, 0.95] | (Cooling rate) |
| $M$ | [10, 15, 20] | (Maximum iterations of unchanged solution) |

We ran 20 randomly seeded iterations of the SA algorithm for each run in the experiment. For each run, the average run time and the average optimality gaps are recorded in Table 7. The run times do not include the time to solve the enumerated SPIPs; the run times only capture how long it takes to solve the upper-level scheduling problem.

The results are ranked first by average optimality gap and second by the average run time. A lower optimality gap is desired as this indicates a higher quality solution. Table 7 displays the top 20 and bottom 20 results returned from this experiment. The parameters that return the highest solution quality and the quickest run time are $N = 40$, $T = 100$, $\lambda = 0.8$, and $M = 10$. The worst set of parameters have low values for $N$ and high values for $T$. The parameters $\lambda$ and $M$ do not significantly affect the solution quality for this experiment.

**Table 7. Experiment on SA Algorithm Hyperparameters**

| $N$ | $T$ | $\lambda$ | $M$ | Avg Time (s) | Avg Opt Gap |
|---|---|---|---|---|---|
| 40 | 100 | 0.8 | 10 | 0.14315 | 0 |
| 40 | 100 | 0.9 | 10 | 0.1486 | 0 |
| 40 | 250 | 0.8 | 15 | 0.192 | 0 |
| 40 | 50 | 0.95 | 15 | 0.1932 | 0 |
| 40 | 250 | 0.8 | 20 | 0.20265 | 0 |
| 40 | 50 | 0.95 | 20 | 0.2192 | 0 |
| 80 | 100 | 0.8 | 10 | 0.2305 | 0 |
| 80 | 50 | 0.95 | 10 | 0.23065 | 0 |
| 80 | 100 | 0.9 | 10 | 0.25795 | 0 |
| 40 | 100 | 0.95 | 20 | 0.2709 | 0 |
| 80 | 100 | 0.95 | 10 | 0.27315 | 0 |
| 80 | 250 | 0.8 | 10 | 0.2846 | 0 |
| 80 | 50 | 0.95 | 15 | 0.297 | 0 |
| 80 | 50 | 0.9 | 15 | 0.29755 | 0 |
| 80 | 250 | 0.9 | 10 | 0.30795 | 0 |
| 80 | 100 | 0.8 | 20 | 0.32985 | 0 |
| 80 | 500 | 0.8 | 10 | 0.3343 | 0 |
| 80 | 250 | 0.8 | 15 | 0.3467 | 0 |
| 80 | 100 | 0.9 | 15 | 0.3582 | 0 |
| 80 | 100 | 0.95 | 15 | 0.3753 | 0 |
| | | | ⋮ | | |
| 10 | 500 | 0.8 | 20 | 0.06265 | 34.3975 |
| 10 | 50 | 0.95 | 15 | 0.084 | 34.6415 |
| 10 | 100 | 0.95 | 10 | 0.06675 | 34.9655 |
| 10 | 100 | 0.9 | 10 | 0.0612 | 39.743 |
| 80 | 500 | 0.95 | 10 | 0.31455 | 43.4365 |
| 10 | 100 | 0.8 | 15 | 0.0484 | 48.0565 |
| 10 | 50 | 0.9 | 10 | 0.05925 | 48.779 |
| 40 | 500 | 0.95 | 10 | 0.18495 | 51.906 |
| 10 | 50 | 0.8 | 20 | 0.0443 | 56.118 |
| 10 | 100 | 0.8 | 20 | 0.0487 | 57.3225 |
| 10 | 250 | 0.8 | 10 | 0.05085 | 57.6565 |
| 10 | 100 | 0.8 | 10 | 0.049 | 58.641 |
| 10 | 500 | 0.95 | 20 | 0.0961 | 69.269 |
| 10 | 500 | 0.8 | 10 | 0.0549 | 74.29 |
| 10 | 250 | 0.9 | 10 | 0.0574 | 74.4195 |
| 10 | 500 | 0.9 | 10 | 0.0612 | 77.56 |
| 10 | 250 | 0.95 | 10 | 0.05665 | 81.005 |
| 10 | 250 | 0.8 | 15 | 0.0555 | 93.0115 |
| 10 | 500 | 0.95 | 15 | 0.06965 | 123.597 |
| 10 | 500 | 0.95 | 10 | 0.04655 | 147.749 |

# V. Conclusions and Recommendations

## 5.1 Conclusions

The nuclear command, control, and communication (NC3) network provides an effective deterrent against adversaries by maintaining robust communication between assets, operators, and decision makers. NC3 must receive a series of upgrades to remain an effective deterrent against evolving threats around the globe. Given a contested network and a portfolio of network upgrades, this research seeks an optimal upgrade schedule to maximize network resilience in support of NC3 modernization. Our research contributes a model that solves a budget constrained network upgrade scheduling problem (NUSP). The NUSP is a defender-attacker-defender tri-level optimization model that achieves fortification via selecting and scheduling network upgrades. We present the tri-level formulation, an exact, solvable solution method, and a simulated annealing (SA) algorithm that solves the upper-level scheduling problem.

We provide insights gained from solving the NUSP using a hypothetical NC3 modernization problem. We compare optimal strategies over four different scenarios, holding the network constant and varying both the attacker's budget and the defender's risk preferences. We further compare these solutions to a myopic (i.e., baseline) solution that ignores interdictor activities and seeks optimal improvement in the (uninterrupted) shortest path. All the scenario solutions showed markedly better resilience scores than the myopic solution, with improvement ranging between 8.2% – 13.1%. Moreover, we determined that the optimal upgrade schedule is influenced by the attacker's budget and defender's risk preferences. Therefore, it is important to accurately characterize these factors to improve solution quality. Lastly, we present an excursion to investigate the best set of SA algorithm parameters tailored to the network instance established in Chapter IV. When implementing the best parameters,

the SA algorithm consistently returns the optimal solution.

## 5.2 Recommendations

Enumerating every attack strategy consumed the majority of computational resources required to solve the NUSP. Therefore, we recommend implementing a quicker shortest path interdiction problem (SPIP) algorithm. In particular, we believe the covering decomposition algorithm of Israeli and Wood (2002), the backwards sampling of Lozano and Smith (2017), and column generation (Desrosiers and Lübbecke, 2005) show significant potential for improving the run time of our exact algorithm.

In addition to algorithmic improvements, future research might also consider stochastic extensions to the NUSP by incorporating uncertainty with regard to attacker budget, interdiction delays, or the source and terminus nodes. We recommend referencing Morton et al. (2007) who apply stochastic programming to uncertain source and terminus nodes for smuggler interdiction.

Lastly, an alternate upper-level scheduling formulation will create a more concise model. It suffices to only decide the upgrade start time rather than identify the phase status of each upgrade (i.e., $w_{ut}$ instead of $w_{upt}$). It is redundant to distinguish between phases since we assume there are no gaps between phases. This improvement should decrease computation time of the upper-level scheduling problem by decreasing the number of decision variables in the formulation.

# Appendix A.

## 1.1 Algorithm to Construct a Solution

**Algorithm 3** Construct Solution

1: Input prioritized($upgrades$), sequence of upgrades in order of priority
2: Input $time$, $budget$, $atk\_dictionary$, from problem instance
3: Initialize $total\_cost = [0, 0, ..., 0]$, zeroized vector of length $time$
4: Initialize $time\_i = 1$
5: **for** $u$ in priority($upgrades$) **do**
6:  **while** $time\_i \leq time$ **do**
7:  **if** $u$ can feasibly start at $time\_i$ **then**
8:  $u_{start} = time\_i$
9:  Update $total\_cost$ with $u_{cost}$
10:  Break while
11:  **else**
12:  $time\_i = time\_i + 1$
13:  **end if**
14:  **end while**
15: **end for**
16: $solution = [u_{start}$ for $u$ in prioritized($upgrades$)]
17: Calculate $\boldsymbol{s}(t)$, the upgrade state configuration for each time period, using $solution$
18: **for** $t = 1$ to $time$ **do**
19:  **if** $\boldsymbol{s}(t) \notin atk\_dictionary$ **then**
20:  Solve Algorithm 1($\boldsymbol{s}(t)$) for $attack\_strategy$
21:  Update $atk\_dictionary$ with key, value pair: $\boldsymbol{s}(t)$, $attack\_strategy$
22:  **end if**
23: **end for**
24: Return $solution$.

On line 20 of Algorithm 3, the optimal attack strategy for each upgrade state configuration in the acquisition timeline is solved by calling Algorithm 1. However, instead of using total enumeration to solve all attack strategies upfront, the algorithm solves each attack strategy as needed. The optimal solution to the attacker's problem is stored in a dictionary for quick access if the same upgrade state configuration is revisited in future iterations. Furthermore, the feasibility check on line 7 checks for two criteria. First, there must be sufficient time left for upgrade $u$ to start at $time\_i$

and finish before *time*. Second, the cost of acquiring $u$ must be within *budget* when added to the total running cost, *total_cost*, of previously budgeted upgrades. As a result, a feasible upgrade strategy solution is returned for a given prioritized upgrade sequence.

## 1.2 Python Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import networkx as nx
import numpy as np
import gurobipy as gp
from gurobipy import GRB
import math
import csv
import random
import re
import matplotlib.pyplot as plt
from matplotlib import pylab
import timeit
import pickle
import sys
from itertools import product

#create class to store the specific info for each upgrade
class Upgrade(object):
    def __init__(self, upgradeID, addArcs, removeArcs, costRD, costPD, costOS, *args, **kwargs):
        #upgrade info array: upgrade ID, RD_yrs, RD_cost, PD_yrs, PD_cost, OS_cost
        self.ID = upgradeID
        self.RD_yrs = len(costRD)
        self.RD_cost = costRD
        self.PD_yrs = len(costPD)
        self.PD_cost = costPD
        self.OS_cost = costOS
        self.addArcs = addArcs
        self.removeArcs = removeArcs

#solution class
class Solution(object):
    def __init__(self, Problem, upgradeOrder, attackDictKeys, startUpgrades, *args, **kwargs):
        self.problem = Problem
        self.key = tuple(upgradeOrder)
        self.operationalUpgrades = attackDictKeys
        self.startUpgrades = startUpgrades

    @property
    def objval(self):
        return sum(self.problem.risk[i] * self.problem.attackDict[self.operationalUpgrades[i]][0]
             for i in range(self.problem.time))
```

```
42
43      #randomly swap order of 2 upgrades and return its solution
44      def nbr2Opt(self):
45          l = [val for val in self.key]
46          i1, i2 = random.sample(range(self.problem.numUpgrades), 2)
47          l[i1], l[i2] = l[i2], l[i1]
48          return self.problem.constructSoln(l)
49
50  #problem class
51  class Problem(object):
52      results = None
53      def __init__(self, G, source, dest, attackerBudget, time, numUpgrades, arcWeight = None,
            arcOperational = None, upgradeObjects = None, RD_budget = None, PD_budget = None,
            OS_budget = None, risk = None, *args, **kwargs):
54
55          #check if path exists between s and t
56          while not nx.has_path(G, source, dest) and dest > source:
57              print("No path from", source, "to", dest,". Setting dest node to", dest-1,".")
58              dest -= 1
59
60          #apply attributes to problem class
61          self.G = G
62          self.arcList = G.edges()
63          self.s = source
64          self.t = dest
65          self.time = time
66          self.numNodes = G.number_of_nodes()
67          self.numUpgrades = numUpgrades
68          self.numArcs = G.number_of_edges()
69          self.attackerBudget = int(attackerBudget)
70          self.attackDict = {}
71          self.allConverged = True
72          self.operationalArcsDict = {}
73
74          #if weight dictionary is not provided, create random weights between 1 and 10
75          if arcWeight:
76              self.arcWeight = arcWeight
77          else:
78              arcWeight = {}
79              arcWeight.update({(i,j): {'weight':random.randint(1, 10)} for i,j in G.edges()})
80              self.arcWeight = arcWeight
81
82          #if operational dictionary is not provided, create random non-operational arcs
83          if arcOperational:
84              self.arcOperational = arcOperational
85          else:
86              arcOperational = {}
87              self.arcOperational = self.createRandomNonOpArcs()
88
89          #update graph attributes with weight and operational dictionaries
90          nx.set_edge_attributes(self.G, self.arcWeight)
91          nx.set_edge_attributes(self.G, self.arcOperational)
92
93          #create random arc cost
```

```
94          costArray, costDict = self.createRandomArcCost()
95          self.costArray = costArray
96          self.costDict = costDict
97
98
99          #if no budget is given, then create budget for each phase
100         if RD_budget and PD_budget and OS_budget:
101             self.budgetRD = RD_budget
102             self.budgetPD = PD_budget
103             self.budgetOS = OS_budget
104         else:
105             self.budgetRD, self.budgetPD, self.budgetOS = self.createRandomBudget()
106
107         #if no upgrade objects are given then create list of upgrade objects
108         if upgradeObjects:
109             self.upgrades = upgradeObjects
110         else:
111             self.upgrades = self.createRandomUpgrades()
112
113         #if no risk is given, create random risk array
114         if risk:
115             self.risk = risk
116         else:
117             self.risk = self.createRandomRisk()
118
119     def solveFortification(self):
120
121         #create params for interdiction problem
122         numUpgrades = self.numUpgrades
123         upgradeStates = [list(key) for key in self.attackDict.keys()]
124         numUpgradeStates = len(upgradeStates)
125         upgradeStatesObjVal = [x[0] for x in self.attackDict.values()]
126         time = self.time
127         risk = self.risk
128         upgradeObjects = self.upgrades
129         defenseBudget = [self.budgetRD, self.budgetPD, self.budgetOS]
130
131         # Create model
132         m = gp.Model("fortification")
133
134         # Create Variables
135         X = m.addVars(numUpgradeStates, time, vtype=GRB.BINARY, name="X")
136         RD = m.addVars(numUpgrades, time, vtype=GRB.BINARY, name="RD")
137         PD = m.addVars(numUpgrades, time, vtype=GRB.BINARY, name="PD")
138         OS = m.addVars(numUpgrades, time, vtype=GRB.BINARY, name="OS")
139
140         # Set objective
141         m.setObjective(gp.quicksum(risk[t] * upgradeStatesObjVal[s] * X[s,t] for s in range(
                numUpgradeStates) for t in range(time)), GRB.MINIMIZE)
142
143         for s, upgrades in enumerate(self.attackDict.keys()):
144             # Set combination (X) of upgrades operational based on the upgrades that are in the OS
                    phase
145             m.addConstrs(X[s, t] * upgrades[u] <= OS[u,t] for u in range(numUpgrades) for t in
```

```
                range(time))
146
147         # Must select an interdiction strategy for each time period, t
148         m.addConstrs((X.sum('*', t) == 1 for t in range(time)), "oneInterdiciton")
149
150         # Only allows one phase at a time per program
151         m.addConstrs(RD[u,t] + PD[u,t] + OS[u,t] <= 1 for u in range(numUpgrades) for t in range(
                time))
152
153         # Add budget constraint for RD, PD, and OS
154         m.addConstrs(gp.quicksum(upgradeObjects[u].RD_cost[0] * RD[u, t] for u in range(
                numUpgrades)) <= defenseBudget[0][t] for t in range(time))
155         m.addConstrs(gp.quicksum(upgradeObjects[u].PD_cost[0] * PD[u, t] for u in range(
                numUpgrades)) <= defenseBudget[1][t] for t in range(time))
156         m.addConstrs(gp.quicksum(upgradeObjects[u].OS_cost * OS[u, t] for u in range(numUpgrades))
                 <= defenseBudget[2][t] for t in range(time))
157
158         # Ensure RD occurs before PD when RD is required (RD_yrs > 0) and ensure PD occurs before
                OS if PD is required
159         # This also ensures the time requirement for RD and PD is met
160         m.addConstrs(PD[u,t] <= gp.quicksum(RD[u,i] for i in range(t+1)) / upgradeObjects[u].
                RD_yrs for u in range(numUpgrades) for t in range(time) if upgradeObjects[u].RD_yrs >
                0)
161         m.addConstrs(OS[u,t] <= gp.quicksum(PD[u,i] for i in range(t+1)) / upgradeObjects[u].
                PD_yrs for u in range(numUpgrades) for t in range(time) if upgradeObjects[u].PD_yrs >
                0)
162
163
164         # Eliminate gaps between phases
165         m.addConstrs(RD[u,t] + PD[u,t] + OS[u,t] >= RD[u,t-1] + PD[u,t-1] + OS[u,t-1] for u in
                range(numUpgrades) for t in range(1, time))
166
167         # Cannot exceed number of RD or PD years for each upgrade (I think this is needed but Im
                not sure)
168         m.addConstrs(RD.sum(u,'*') <= upgradeObjects[u].RD_yrs for u in range(numUpgrades))
169         m.addConstrs(PD.sum(u,'*') <= upgradeObjects[u].PD_yrs for u in range(numUpgrades))
170
171         # Solve model
172         m.optimize()
173         results = {}
174
175         # Return results
176         if m.solCount > 0:
177             print("Objective value:", m.objVal)
178             #create dictionary of results
179             for v in m.getVars():
180                 varArray = tuple(int(x.strip("[]")) for x in v.varName[v.varName.find("["):].split
                        (','))
181                 varN = "".join(re.split("[^a-zA-Z]*", v.varName))
182                 varValue = float(v.x)
183                 varKey = (varN,) + varArray
184                 results.update({varKey: varValue})
185
186         return results, m.objVal
```

```
187
188     #create random non operational arcs based on highest degree nodes within shortest paths
189     def createRandomNonOpArcs(self):
190         arcOperational = {}
191         subG = self.G.subgraph(self.G.nodes).copy()
192         removedArcs = None
193         removedNode = None
194
195         #sort by degree of nodes in self.G
196         while nx.has_path(subG, self.s, self.t):
197             shortestNodePath = nx.dijkstra_path(subG, self.s, self.t, weight = "weight")
198             sortedNodes = sorted(self.G.degree(shortestNodePath), key=lambda x: x[1])
199
200             #continue to remove nodes if sorted list has s or t as highest degree nodes
201             while sortedNodes and (sortedNodes[-1][0] == self.s or sortedNodes[-1][0] == self.t):
202                 sortedNodes.pop()[0]
203
204             #take the in/out edges from removed node and add to removedArcs
205             if sortedNodes:
206                 removedNode = sortedNodes.pop()[0]
207                 removedArcs = set(self.G.in_edges(removedNode)) | set(self.G.out_edges(removedNode
                        ))
208                 subG.remove_edges_from(removedArcs)
209             #if no node exists between the shortest path, remove this path and continue
210             else:
211                 removedArcs = getArcs(shortestNodePath)
212                 subG.remove_edges_from(removedArcs)
213
214
215         #add back in the last set of removedArcs so a path exists between s and t
216         subG.add_edges_from(removedArcs)
217
218         #get set of non operational arcs
219         nonOpArcs = set(self.G.edges) - set(subG.edges)
220
221         #create dictionary of operational arcs
222         #count non op arcs
223         countNonOp = 0
224         for i,j in self.G.edges:
225             if (i,j) in nonOpArcs:
226                 arcOperational.update({(i,j): {'o': 0}})
227                 countNonOp += 1
228             else:
229                 arcOperational.update({(i,j): {'o': 1}})
230         if countNonOp < self.numUpgrades:
231             message = str(countNonOp) + " non-operational arcs were randomly generated. This is
                    less than the number of upgrades. Please decrease number of upgrades or increase
                    network size."
232             sys.exit(message)
233
234         #remove subgraph
235         subG.clear()
236         return arcOperational
237
```

```
    #random risk array
    def createRandomRisk(self):
        return list(np.random.randint(1,10,self.time))

    #randomly assign non operational arcs to an upgrade
    def createRandomUpgrades(self):
        #get list of arcs and nodes not yet operational
        nonOpArcs = [(i,j) for i,j in self.arcList if self.arcOperational[(i,j)]['o'] == 0]
        nonOpG = nx.DiGraph(nonOpArcs)
        nonOpTrees = sorted(nx.weakly_connected_components(nonOpG), key=len, reverse=True)

        #create dictionary of edges for each non-op tree
        treeDict = {}
        for treeID, tree in enumerate(nonOpTrees):
            treeDict.update({treeID: set(nonOpG.edges(tree))})

        #create list of lists for arcs to be added or removed for each upgrade
        #note, no remove arcs exist for random upgrades
        addArcs = []
        removeArcs = []
        for iter in range(self.numUpgrades):
            addArcs.append([])
            removeArcs.append([])

        #if there are more non-operational, upgradeable components (trees) than upgrades
        #then assign each tree to an upgrade
        if len(nonOpTrees) >= self.numUpgrades:
            #add the arcs from each tree to an upgrade
            for treeID,tree in enumerate(nonOpTrees):
                arcsInTree = treeDict[treeID]
                addArcs[treeID % self.numUpgrades].extend(arcsInTree)

        #not enough trees to be divided up, instead divide arcs up
        else:
            splitArcs = np.array_split(nonOpArcs, self.numUpgrades)
            for idx, split in enumerate(splitArcs):
                arcsInSplit = [(u,v) for u,v in split]
                addArcs[idx].extend(arcsInSplit)

        #enumerate through each upgrade and create an upgrade object
        #cost is scaled based on number of arcs in each upgrade
        upgradeObjects = []
        maxNumArc = max(len(a) for a in addArcs)
        minNumArc = min(len(a) for a in addArcs)
        minRD, maxRD = 100, 200
        minPD, maxPD = 120, 250

        for upgradeID in range(self.numUpgrades):
            #determine PD and RD cost
            if minNumArc == maxNumArc:
                rdCost = minRD
                pdCost = minPD
            else:
                rdCost = int(((len(addArcs[upgradeID]) - minNumArc) / (maxNumArc - minNumArc)) * (
```

```
                    maxRD - minRD) + minRD)
292             pdCost = int(((len(addArcs[upgradeID]) - minNumArc) / (maxNumArc - minNumArc)) * (
                    maxPD - minPD) + minPD)
293             rd = [rdCost] * random.randint(1, 3)
294             pd = [pdCost] * random.randint(1, 3)
295             os = np.random.randint(150, 220)
296
297             #create list of upgrade objects
298             upgradeObjects.append(Upgrade(upgradeID, addArcs[upgradeID], removeArcs[upgradeID], rd
                    , pd, os))
299
300         #clear graphs
301         nonOpG.clear()
302
303         #return all upgrade objects
304         return upgradeObjects
305
306     #create random budget
307     def createRandomBudget(self):
308         #(this isnt actually random right now)
309         budgRD = [200] * self.time
310         budgPD = [250] * self.time
311         budgOS = [150 * self.numUpgrades] * self.time
312         return budgRD, budgPD, budgOS
313
314
315     #create random cost function for problem
316     def createRandomArcCost(self):
317         #ensure resources is int (round down if not)
318         costDict = {}
319
320         #create cost function with cost array
321         #create cost dictionary that maps an arc to the index in the cost array
322         costArray = np.random.randint(0, 15, size = (self.numArcs, self.attackerBudget))
323         costArray = np.sort(costArray, axis=1)
324         costArray = costArray.tolist()
325
326         costDict.update({arc: idx for idx, arc in enumerate(self.arcList)})
327         #set some of the arcs to be severed
328         #costArray = [[val if val < 15 else 999 for val in row] for row in costArray]
329
330         return (costArray, costDict)
331
332     #update cost array to only include arcs (rows in cost array) visible to interdictor
333     def updateCostArray(self, Y_Hat_ArcList):
334         costDict = {}
335         costArray = [self.costArray[self.costDict[arc]] for arc in Y_Hat_ArcList]
336         costDict.update({arc: i for i, arc in enumerate(Y_Hat_ArcList)})
337         return costArray, costDict
338
339     #solve the shortest path interdiction problem and track the interdiction strategy
340     def getSPIP(self, key):
341         maxItrs = 40        #THIS MAY NEED INCREASED (depends on prob size)
342         backupCounter = 0
```

```
343         tol = .001
344         LB = 0
345         UB = float('inf')
346 
347         #initialize dictionaries and lists
348         interdictionDict = {}
349         Y_Hat_ArcList = []
350         costDict = {}
351 
352         #update graph with specific upgrade instance
353         self.updateGraph(list(key))
354         #get operational edge list
355         opEdges = [(u,v) for (u,v) in self.arcList if self.G[u][v]['o'] == 1]
356         #create subgraph of only operational edges
357         opG = nx.DiGraph(self.G.edge_subgraph(opEdges))
358 
359         while not math.isclose(UB, LB, rel_tol = tol) and backupCounter < maxItrs:
360             #get sugraph using shortest path
361             Y_Hat, Y_Hat_ArcList = updateYHat(self, Y_Hat_ArcList, opG)
362             #pair down cost array to only arcs visited (remove all other rows in array not visited
                    )
363             costArray, costDict = self.updateCostArray(Y_Hat_ArcList)
364             #reset weight attribute
365             nx.set_edge_attributes(opG, self.arcWeight)
366             #solve new interdiction strategy
367             interdictionDict, UB = solveSPIP(self.G, Y_Hat, Y_Hat_ArcList, self.attackerBudget,
                     costArray, costDict)
368             #set arc weights
369             for arc in interdictionDict.keys():
370                 opG[arc[0]][arc[1]]['weight'] += interdictionDict[arc]
371             #set lower bound
372             LB = nx.shortest_path_length(opG, self.s, self.t, 'weight')
373             backupCounter += 1
374         #update operational arcs dict with all operational edges, and Y_hat arc list of edges
375         self.operationalArcsDict.update({key: (set(opEdges), Y_Hat_ArcList)})
376 
377         checkCounterBool = bool(backupCounter < maxItrs)
378         if not checkCounterBool:
379             self.allConverged = False
380 
381         #reset the operational arcs to initial state
382         nx.set_edge_attributes(self.G, self.arcOperational)
383         #remove graph
384         opG.clear()
385 
386         return LB, interdictionDict, checkCounterBool
387 
388     #update operational arcs in graph based on the upgrade instance given
389     def updateGraph(self, upgradeInstance):
390         #determine set of operational and non operational arcs
391         operationalArcs = []
392         nonOperationalArcs = []
393         for idx, upgradeBin in enumerate(upgradeInstance):
394             if upgradeBin == 1:
```

```
                operationalArcs += [arc for arc in self.upgrades[idx].addArcs]
            #add back in the arcs to be removed if the upgrade that removes arc is not implemented
            if upgradeBin == 1:
                nonOperationalArcs += [arc for arc in self.upgrades[idx].removeArcs]

        #set arcs in graph to operational or non operational depending on the upgrades implemented
        for (u,v) in set(operationalArcs).union(set(nonOperationalArcs)):
            if (u,v) in set(operationalArcs) - set(nonOperationalArcs):
                self.G[u][v]['o'] = 1
            else:
                self.G[u][v]['o'] = 0
        return


    #construct solution for heuristic
    def constructSoln(self, upgradeOrder):
        #initialize variables
        sumRD = [0] * self.time
        sumPD = [0] * self.time
        sumOS = [0] * self.time
        startUpgradeDict = {}
        timeIndex = 0
        timeLeft = self.time - timeIndex - 1

        #iterate over every upgrade and start its acquisition when feasbile in the proper upgrade order
        for u in upgradeOrder:
            #keep indexing over time period, timeIndex, until upgrade u can start its acquisition
            while timeIndex < self.time:
                startRDIdx = timeIndex
                startPDIdx = startRDIdx + self.upgrades[u].RD_yrs
                startOSIdx = startPDIdx + self.upgrades[u].PD_yrs
                #if feasible, set start and break
                if timeLeft >= (self.upgrades[u].RD_yrs + self.upgrades[u].PD_yrs) and \
                    all(cost + sumRD[startRDIdx + i] <= self.budgetRD[startRDIdx + i] for i, cost in enumerate(self.upgrades[u].RD_cost)) and \
                    all(cost + sumPD[startPDIdx + i] <= self.budgetPD[startPDIdx + i] for i, cost in enumerate(self.upgrades[u].PD_cost)) and \
                    all(self.upgrades[u].OS_cost + sumOS[startOSIdx + i] <= self.budgetOS[startOSIdx + i] for i in range(self.time - startOSIdx)):

                    #update the current cost of acquisitions
                    for i, cost in enumerate(self.upgrades[u].RD_cost):
                        sumRD[startRDIdx + i] += cost
                    for i, cost in enumerate(self.upgrades[u].PD_cost):
                        sumPD[startPDIdx + i] += cost
                    for i in range(self.time - startOSIdx):
                        sumOS[startOSIdx + i] += self.upgrades[u].OS_cost

                    #record the starting time period of the aquisiton upgrade for every upgrade and break from loop
                    startUpgradeDict.update({u: timeIndex})
                    break

```

```
444                 #increase the time index and try to start upgrade u at timeIndex
445                 timeIndex += 1
446                 timeLeft -= 1
447 
448         #get optimal attack strategy for each time period based on startUpgradeDict
449         attackDictKeys = [None] * self.time
450         for t in range(self.time):
451             attackDictKeys[t] = tuple([0 if (u not in startUpgradeDict.keys() or t < (
                    startUpgradeDict[u] + self.upgrades[u].RD_yrs + self.upgrades[u].PD_yrs)) else 1
                    for u in range(self.numUpgrades)])
452 
453         #if key does not exist in attackDict, solve for optimal attack strategy and add to
                dictionary
454         for key in attackDictKeys:
455             if key not in self.attackDict.keys():
456                 #solve spip if it has not already been solved
457                 LB, interdictionDict, checkCounterBool = self.getSPIP(key)
458                 self.attackDict.update({key:(LB, interdictionDict)})
459 
460         #store soln in dict
461         soln = Solution(self, upgradeOrder, attackDictKeys, startUpgradeDict)
462 
463         #return solution
464         return soln
465 
466     #randomly shuffle upgrade order and reutrn its solution
467     def solnRandom(self):
468         return self.constructSoln(random.sample(list(range(self.numUpgrades)), self.numUpgrades))
469 
470     #solve the problem via Simulated Annealing heuristic
471     def SA(self, sa_run):
472         innerIters = sa_run[0]
473         temp = sa_run[1]
474         cooling = sa_run[2]
475         maxUnchangedIters = sa_run[3]
476         seed = sa_run[4]
477 
478         #Initialize
479         np.random.seed(seed)
480         Soln = self.solnRandom()
481         BestSoFar = Soln
482         minTemp = 1
483         count = 0
484         solnDict = {Soln.key:Soln.objval}
485 
486         while temp >= minTemp:  ## The main loop
487             prevBestObjVal = BestSoFar.objval
488             #Soln = BestSoFar
489             for itr in range(innerIters):
490                 Candidate = Soln.nbr2Opt()
491                 d = Candidate.objval - Soln.objval
492                 if d <= 0:
493                     Soln = Candidate
494                     #new solution is better than previous best, reset count and set the new best
```

```
                        soln
                    if Soln.objval <= BestSoFar.objval:
                        solnDict.update({Soln.key: Soln.objval})
                        if Soln.objval < BestSoFar.objval:
                            count = 0
                            BestSoFar = Soln
                else:
                    r = np.random.random()
                    if r < math.exp(-d/temp):
                        Soln = Candidate
            #check if best so far is equal to previous best
            if prevBestObjVal == BestSoFar.objval:
                count += 1
                #if no change after M iterations, exit loop
                if count == maxUnchangedIters:
                    return BestSoFar, solnDict
            temp = cooling*temp

        return BestSoFar, solnDict

def solveSPIP(G, Y_Hat, Y_Hat_ArcList, numResources, costArray, costDict):

    numArcs = len(Y_Hat_ArcList)
    interdictionDict = {}
    try:
        # Create a new model
        m = gp.Model("spip")
        # Create variables
        x = m.addVars(numArcs, numResources, vtype=GRB.BINARY, name="x")
        z = m.addVar(vtype=GRB.CONTINUOUS, name="z")
        # Set objective
        m.setObjective(z, GRB.MAXIMIZE)
        #interdiction within resources (b)
        m.addConstr(gp.quicksum(x.sum('*', j) * (j+1) for j in range(numResources)) <=
             numResources, "resources")
        #max of one interdiction strategy per arc
        m.addConstrs(x.sum(i, '*') <= 1 for i in range(numArcs))
        #add constraints for each path in Y_hat
        for idx, path in enumerate(Y_Hat):
            indexList = []
            indexList += [costDict[arc] for arc in path]
            #benders cuts
            m.addConstr((getPathLength(G, path) + gp.quicksum(x[i,j] * costArray[i][j] for i in
                 indexList for j in range(numResources)) >= z), "zConstrs")

        # Optimize model
        m.optimize()
        #if unbounded, return infinity for obj value
        if m.solCount == 0:
            m.objVal = float('inf')

        #save interdiction info
        else:
            #add info to dict if the DV (x) is greater than 0.5.
```

```
            #I choose 0.5 here since some rounding error can occur and x may equal 0.0001 or
                0.9999.
            #in which case, using 0.5 to delineate between ~0 and ~1 is best
            interdictionDict.update({Y_Hat_ArcList[i]: costArray[i][j] for (i,j) in x.keys() if x
                [(i,j)].x > 0.5})

    except gp.GurobiError as e:
        print('Error code ' + str(e.errno) + ': ' + str(e))

    except AttributeError:
        print('Encountered an attribute error')

    return (interdictionDict, m.objVal)

#get the length of the path
def getPathLength(G, path):
    return sum(G[u][v]['weight'] for (u,v) in path)

#create list of edges from a given path
def getArcs(nodePath):
    arcList = [(nodePath[idx], nodePath[idx+1]) for idx in range(len(nodePath)-1) if len(nodePath)
        > 1]
    return arcList

#find the shortest path of graph, post-intediction
#append the arcs from new shortest path to the arc list
#create a subgraph from the arc list and find all simple s-t paths in subgraph
#the collection of s-t paths is Y_hat
def updateYHat(problem, arcList, G):

    #Y_Hat is list of paths currently in Y_hat_graph
    shortestPath = nx.dijkstra_path(G, problem.s, problem.t, weight = 'weight')
    shortestPathArcs = getArcs(shortestPath)
    arcList = list(set(arcList).union(set(shortestPathArcs)))
    Y_hat_graph = nx.DiGraph(G.edge_subgraph(arcList))
    allPaths = nx.all_simple_edge_paths(Y_hat_graph, problem.s, problem.t)

    #get list of every path from all simple s-t path generator object
    Y_hat = []
    for path in allPaths:
        Y_hat.append(path)
    #clear out the graph
    Y_hat_graph.clear()

    return (Y_hat, arcList)

#enumerate all upgrade state combinatiions (0, 0, ... n) to (1, 1, ... n) using recursion
def enumerateInstances(problem, upgradeInstance, i):
    if i == problem.numUpgrades:
        problem.attackDict.update({tuple(upgradeInstance): None})
        return
    upgradeInstance[i] = 1
    enumerateInstances(problem, upgradeInstance, i + 1)
    upgradeInstance[i] = 0
```

```
597     enumerateInstances(problem, upgradeInstance, i + 1)
598 
599 #create a loop to iteratively solve multiple instances of a problem using multiple processes
600 def heuristicLoop(problem, sa_run):
601     #start timer
602     startTime = timeit.default_timer()
603 
604     #solve the problem with SA heuristic, return the best solution and dictionary of all found
            best solns
605     bestSoln, solnsDict = problem.SA(sa_run)
606     for (timeline, objVal) in solnsDict.items():
607         print(timeline, objVal)
608 
609     #end timer
610     endTime = timeit.default_timer()
611 
612     timeDiff = round(endTime-startTime,3)
613     #print best soln
614     print("Objective value: ", bestSoln.objval)
615     print("All attacker problems converged:", problem.allConverged)
616     print("Upgrade priority of best solution:", bestSoln.key)
617     print("Timeline to start upgrades:", bestSoln.startUpgrades)
618     print("Time to solve:", timeDiff, "seconds")
619     print("SA run", sa_run, "\n")
620     return bestSoln.objval, timeDiff, solnsDict
621 
622 #create a loop to iteratively solve multiple instances of a problem using multiple processes
623 def optimizationLoop(problem):
624     #start timer
625     startTime = timeit.default_timer()
626 
627     upgradeInstance = [None] * problem.numUpgrades
628     enumerateInstances(problem, upgradeInstance, 0)
629     #for every combination of upgrade strategies, solve the optimal attack strategy
630     for key in problem.attackDict.keys():
631         LB, interdictionDict, checkCounterBool = problem.getSPIP(key)
632         problem.attackDict.update({key:(LB, interdictionDict)})
633 
634     #solve the upper level fortification problem
635     results, objVal = problem.solveFortification()
636 
637     #end timer
638     endTime = timeit.default_timer()
639 
640     #get upgrade order
641     upgradeOrder = []
642     for u in range(problem.numUpgrades):
643         timelineRD = [int(results['RD', u, t]) for t in range(problem.time)]
644         #if upgrade isn't selected for upgrade, then set its time to 99
645         try:
646             t = timelineRD.index(1)
647         except:
648             t = 99
649         upgradeOrder.append((u,t))
```

```
    upgradeOrder.sort(key=lambda x: x[1])

    timeDiff = round(endTime-startTime,3)
    #print results
    print("All attacker problems converged:", problem.allConverged)
    print("Timeline to start upgrades (upgrade, time):", upgradeOrder)
    print("Time to solve:", timeDiff, "seconds\n")

    #print timeline of all phases
    optimalTimeline(problem, results)

    return objVal, timeDiff, results

#main
if __name__ == "__main__":
    #call gurobi.env to supress gurobi output
    with gp.Env(empty=True) as env:

        #initialize seeds and attacker budget
        seed = 4
        np.random.seed(seed)
        random.seed(seed)

        #create problem instances
        #get data from csv file
        filename = "Rome.csv"
        attackerBudget = 40
        numUpgrades = 10
        time = 20
        risk = [1]*10 + [10]*10

        #SA hyperparams (add more values to each list for full factorial DOE)
        innerIters = [40]
        temp = [100]
        cooling = [.8]
        maxUnchangedIters = [10]
        seedList = [seed]
        DOE_SA = list(product(innerIters, temp, cooling, maxUnchangedIters, seedList))

        with open(filename, newline='', encoding='utf-8-sig') as csvfile:
            reader = list(csv.reader(csvfile, dialect = 'excel'))
            data = np.array(reader)
            csvfile.close()

        #turn data into usable format, (u, v) node pair and w = weight
        uList = [int(u) for u in data[1:,0]]
        vList = [int(v) for v in data[1:,1]]
        wList = [float(w)/100 for w in data[1:,2]]
        arcList = list(zip(uList, vList))

        #Create graph with the list of arcs and set attributes for initial state of the graph
        G = nx.DiGraph(arcList)
        arcWeight = {}
        arcWeight.update({(i,j): {'weight':wList[idx]} for idx, (i,j) in enumerate(arcList)})
```

```
        nx.set_edge_attributes(G, arcWeight)
        source = 116
        dest = 2042

        #create problem instance
        prob = Problem(G, source, dest, attackerBudget, time, numUpgrades, arcWeight = arcWeight,
             risk = risk)
        print("Attacker budget:", prob.attackerBudget)
        print("Risk:", prob.risk)

        #solve with optimization
        print("-----opt-----")
        optObjVal, optTime, results = optimizationLoop(prob)

        #solve with heuristic
        solnsDictList= []
        for run in DOE_SA:
            print("-----heur-----")
            prob.attackDict = {}
            heurObjVal, heurTime, solnsDict = heuristicLoop(prob,run)
            solnsDictList.append(solnsDict)

        #save data for future use
        with open('Solution_Data.pkl', 'wb') as f:
            pickle.dump([prob, results, solnsDictList], f)
            f.close()
```

# Bibliography


Alderson, D. L., Brown, G. G., Carlyle, W. M. and Cox Jr, L. A. (2013), 'Sometimes there is no 'most-vital' arc: Assessing and improving the operational resilience of systems', *Military Operations Research* **18**(1), 21–37.

Brown, G., Carlyle, M., Salmeron, J. and Wood, K. (2005), 'Analyzing the vulnerability of critical infrastructure to attack and planning defenses', *INFORMS TutORials in Operations Research* pp. 102–123.

Brown, G., Carlyle, M., Salmerón, J. and Wood, R. (2006), 'Defending critical infrastructure', *Interfaces* **36**, 530–544.

Cantillo, V., Macea, L. F. and Jaller, M. (2019), 'Assessing vulnerability of transportation networks for disaster response operations', *Networks & Spatial Economics* **19**(1), 243–273.

Chow, J., Regan, A., Ranaiefar, F. and Arkhipov, D. (2011), 'A network option portfolio management framework for adaptive transportation planning', *Transportation Research Part A: Policy and Practice* **45**, 765–778.

Department of Defense, U. S. (2006), *Deterrence Operations Joint Operating Concept*, Washington DC: GPO.

Deptula, D., LaPlante, W. and Haddock, R. (2019), 'Modernizing US nuclear command, control, and communications'.

Desrosiers, J. and Lübbecke, M. (2005), 'A primer in column generation', *Operations Research* **53**(6), 1007–1023.

Du, Q., Kishi, K., Aiura, N. and Nakatsuji, T. (2014), 'Vulnerability scanning methodology applied to multiple logistics transport networks', *Journal of the Transportation Research Board* pp. 96–104.

Garey, M. R., Johnson, D. S. and Sethi, R. (1976), 'The complexity of flowshop and jobshop scheduling', *Mathematics of Operations Research* **1**(2), 117–129.

Ghorbani-Renani, N., González, A. D., Barker, K. and Morshedlou, N. (2020), 'Protection-interdiction-restoration: Tri-level optimization for enhancing interdependent network resilience', *Reliability Engineering & System Safety* **199**.

Hagberg, A. A., Schult, D. A. and Swart, P. J. (2008), Exploring network structure, dynamics, and function using NetworkX, *in* 'Proceedings of the 7th Python in Science Conference', pp. 11–15.

Israeli, E. and Wood, R. K. (2002), 'Shortest-path network interdiction', *Networks: An International Journal* **40**(2), 97–111.

Kang, Z., Kumar, A., Harrison, T. and Yen, J. (2011), 'Analyzing the resilience of complex supply network topologies against random and targeted disruptions', *IEEE Systems Journal* **5**(1), 28–39.

Kirkpatrick, S., Gelatt, C. and Vecchi, M. (1983), 'Optimization by simulated annealing', *Science* **220**, 671–680.

Lazzaro, G. L. (2016), Tri-level optimization algorithms for solving defender-attacker-defender network models, PhD thesis, Naval Postgraduate School.

Lozano, L. and Smith, J. (2017), 'A backward sampling framework for interdiction problems with fortification', *INFORMS Journal on Computing* **29**, 123–139.

Maggioni, F., Potra, F. A. and Bertocchi, M. (2017), 'A scenario-based framework for supply planning under uncertainty: stochastic programming versus robust optimization approaches', *Computational Management Science* **14**(1), 5.

Morton, D. P., Pan, F. and Saeger, K. J. (2007), 'Models for nuclear smuggling interdiction', *IIE Transactions* **39**(1), 3–14.

Office of the Secretary of Defense, U. S. (2018), *Nuclear Posture Review*, Washington DC: GPO.

Smith, C. J., Prince, M. and Geunes, J. (2013), 'Modern network interdiction problems and algorithms', *Handbook of Combinatorial Optimization* pp. 1949–1987.

Storchi, G., Dell'Olmo, P. and Gentili, M. (1999), 'Directed road network of the city of Rome'. Retrieved January 4, 2021, from `http://users.diag.uniroma1.it/challenge9/data/rome/rome99.gr`.

Van Laarhoven, P. J. M., Aarts, E. H. L. and Lenstra, J. K. (1992), 'Job shop scheduling by simulated annealing', *Operations Research* **40**(1), 113–125.

von Stackelberg, H. (1952), *The Theory of the Market Economy*, Oxford University Press.

Walker, B., Holling, C., Carpenter, S. and Kinzig, A. (2003), 'Resilience, adaptability and transformability in social-ecological systems', *Ecology and Society* **9**(2).

Wood, K. R. (1993), 'Deterministic network interdiction', *Mathematical and Computer Modelling* **17**(2), 1–18.

Zhao, K., Scheibe, K., Blackhurst, J. and Kumar, A. (2019), 'Supply chain network robustness against disruptions: Topological analysis, measurement, and optimization', *IEEE Transactions on Engineering Management* **66**(1), 127–139.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704–0188*



| 1. REPORT DATE *(DD–MM–YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From — To)* |
|---|---|---|
| 25–03–2021 | Master's Thesis | Sep 2019 — Mar 2021 |

**4. TITLE AND SUBTITLE**

Strategic Upgrade Scheduling and Portfolio Management to Fortify Contested Networks

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

Anderson, Justin L., Capt, USAF

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

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

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

**8. PERFORMING ORGANIZATION REPORT NUMBER**

AFIT-ENS-MS-21-M-141

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

AF/A10
Col John E. Dines, USAF
1488 Air Force, Suite 4E240
Pentagon, Washington D.C. 20330
john.e.dines.mil@mail.mil

**10. SPONSOR/MONITOR'S ACRONYM(S)**

AF/A10

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

**12. DISTRIBUTION / AVAILABILITY STATEMENT**

Distribution Statement A. Approved for Public Release; Distribution Unlimited.

**13. SUPPLEMENTARY NOTES**

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

**14. ABSTRACT**

The nuclear command, control, and communication (NC3) network provides an effective deterrent against adversaries by maintaining robust communication between assets, operators, and decision makers. NC3 must receive a series of upgrades to remain an effective deterrent against evolving threats around the globe. Given a contested network and a portfolio of network upgrades, this research seeks an optimal upgrade schedule to maximize network resilience in support of NC3 modernization. Our research contributes a model that solves a budget-constrained network upgrade scheduling problem (NUSP). The NUSP is a defender-attacker-defender tri-level optimization model that achieves fortification via selecting and scheduling network upgrades. The lower-level defender problem is the shortest path problem, the mid-level attacker problem is the shortest path interdiction problem (SPIP), and the upper-level defender's problem is the NUSP. We present an exact, solvable solution method and propose a simulated annealing (SA) heuristic. The formulation is applied to a large network instance. Using four scenarios, we investigate the effects of attacker budget and risk profiles on the optimal upgrade schedule. Results indicate that both attacker budget and risk profile affect the optimal upgrade strategy. Moreover, the optimal upgrade schedule provides greater network resilience than the myopic (i.e., baseline) upgrade schedule.

**15. SUBJECT TERMS**

Trilevel programming, bilevel programming, networks, robust optimization, game theory, scheduling, shortest path problem, shortest path interdiction problem, network upgrade scheduling problem, nuclear command control and communications

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Lt Col Timothy W. Holzmann, AFIT/ENS |
| U | U | U | UU | 77 | 19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636; Timothy.Holzmann@afit.edu |