Architecture-Based Graceful Degradation for Cybersecurity

Ryan R. Wagner

CMU-S3D-25-104
May 2025

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Prof. David Garlan, Co-Chair
Prof. Matt Fredrikson, Co-Chair
Prof. Jonathan Aldrich
LTG Peter Kind, USA, Ret.

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering

This work was sponsored by the National Security Agency (NSA) under awards no. H9823014C0140 and H9823018D0008, the Defense Advanced Research Projects Agency (DARPA) under awards no. FA87501520277 and FA87501620042, the National Science Foundation (NSF) under awards no. CNS1704542 and CNS1943016, the Office of Naval Research (ONR) under award no. N000141712899, Sandia National Laboratories under award no. PO1988472, Intelligent Automation under award no. 24331, the Portuguese Science and Technology Foundation under award no. 1031319, and Cylab under awards no. 5003308 and 5007072. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSA, DARPA, NSF, ONR, Sandia National Laboratories, Intelligent Automation, the Portugues Science and Technology Foundation, Cylab, the U.S. government or any other entity.

Keywords: Software architecture, cybersecurity, security, resilience, graceful degradation

Abstract

Successful attacks are nearly inevitable as sophisticated threat actors are committed to inflicting damage, leaving digital and physical destruction in their wakes. As defenders recognize the inevitability of successful attacks, they must change their defense paradigms from only preventing attacks to also weathering the attacks that penetrate first-line defenses. Instead, the systems' abilities to provide functionality should be minimally disrupted while simultaneously containing an attacker. The engineering challenge is to build and operate systems that are resilient to attack, able to adapt to trade off some functionality to preserve trust in more-critical functionality. We refer to this concept as graceful degradation. Defenders would be in a far better position to address the increasingly dire situation confronting them if they had a method and tool to support graceful degradation. However, this requires the ability to reason despite uncertainties at architecture and design time and at run time. Automation can be supported by formal modeling of systems, but it must not be labor-intensive. We propose and develop an approach that directly addresses these challenges. We can architect and operate systems that are better able to weather attacks by automating the evaluation of systems' security properties to enable effective automated graceful degradation of systems in the presence of uncertainty through an approach of formally modeling systems and system behavior at an architectural level of abstraction to explore hypothetical attacks and the systems' abilities to respond. We describe our approach and provide tooling to demonstrate our concept.

Dedication

To my dog Dora, who passed away during the pandemic. She was most loving, loyal companion and study-buddy I could have ever hoped for. And to my family: I know I have some visits that I owe you all!

Acknowledgments

I first want to thank my advisors David and Matt for their time, patience, and expertise. They really know their stuff and guided me through this experience in a manner for which I will always be grateful. Thank you, Jonathan, for all your feedback. General Kind, your real-world experience and insight are invaluable guides to making sure I am trying to solve the right problems. Bradley, Connie, Alisha: I do not know how I could have done this without your help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A digital storm is brewing as important systems become smarter, more interconnected, and increasingly complicated. Successful attacks are nearly inevitable as sophisticated threat actors are committed to inflicting damage, leaving digital and physical destruction in their wakes. As defenders recognize the inevitability of successful attacks, they must change their defense paradigms from only preventing attacks to also weathering the attacks that penetrate first-line defenses.

When under attack, it is often suboptimal to shut down an entire system [1]. This type of complete shutdown is particularly problematic for systems with reliability or availability requirements, such as satellites, electrical utilities, and military operations, as well as, more generally, for commercial systems, which must maintain high availability for business reasons. Instead, the systems' abilities to provide functionality should be minimally disrupted while simultaneously containing an attacker. The engineering challenge is to build and operate systems that are resilient to attack, able to adapt to trade off some functionality to preserve trust in more-critical functionality. We refer to this concept as *graceful degradation*.

A variety of causes conspire to make it difficult to create and operate systems that support graceful degradation for security. To be most effective, graceful degradation capabilities must be designed into the system rather than bolted on during system operations, and graceful degradation must continue to be a first class concern through run time operations. Unfortunately, current design methods and system representations make it difficult to reason methodically about graceful degradation during design. Existing methods and representations are informal and manual, and they represent facets of the system that are difficult to integrate coherently. This problem extends into implementation and operations, when the original design artifacts remain and new, incompatible representations are added.

Further exacerbating the problem, there are a number of known unknowns that are relevant to graceful degradation, but difficult to incorporate into existing approaches. For example, at design time, many implementation details are not known to the defender. Even at run time, the defender cannot possibly know all the latent vulnerabilities in his system. Throughout the system lifecycle, a sophisticated adversary may have Tactics, Techniques, and Procedures (TTPs) that are difficult to predict; she may also deliberately make herself difficult to detect by producing few observable artifacts. When dealing with such uncertainties regarding attackers and systems, it might seem like a hopeless task to try to estimate risk. Explicitly designing for graceful degradation for security – and successfully executing it in operations – is an extraordinarily challenging task, and current methods and tooling fall short in providing support to architects and administrators that wish to anticipate and respond to attacks.

Defenders would be in a far better position to address the increasingly dire situation confronting them if they had a method and tool to support graceful degradation that balances functionality with security risks. Ideally, the method and tool should integrate with current processes, leverage existing sources of data while not requiring unobtainable information, and it should not be onerous to use. Further, the method/tool outputs must be trusted by system architects to guide architecture and design, and the outputs must be trusted by system administrators to provide accurate attack response guidance at run time when time is precious and there is little room for error. For the best results, an integrated approach must work seamlessly from the architecture and design phase of the system lifecycle – producing design guidance to increase system resilience against security threats – to the operations phase – producing Courses of Action (COAs) or Automated Courses of Action (ACOAs) in response to updated information about an attack.

In this thesis, we propose and develop an approach that directly addresses these challenges. We integrate and extend features of existing approaches with a first class treatment of uncertainty and a formal rigor that allows designers and operators to reason quantitatively and in an automated fashion about graceful degradation in response to attacks. As part of this thesis, we represent and integrate the disparate stakeholder concerns relevant to graceful degradation for security through formal modeling at an architectural level of abstraction. This approach captures the concerns relevant to graceful degradation for security at both the early stage design – when few implementation details are known – and in operations – when high-level models are needed to reason effectively about system-level responses to an ongoing security attack.

An architectural level of abstraction is defined as the "gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives" [2]. Crucially, the architectural level of abstraction is the level at which trade-offs can be evaluated among degraded architectures. At design time, this level of abstraction is available for reasoning even when implementation details are yet to be defined; systems can be architected to build in an ability to degrade *gracefully* while under attack. At run time, an architectural level of abstraction contains the key information for making appropriate functional trade-offs to manage risk from ongoing attacks.

At both architecture/design time and during run time, our approach evaluates risk by exploring the space of hypothetical attack traces on architectures to estimate the resilience of architectural alternatives to various attack scenarios. We focus on the impacts of attack traces to *data flows*, and the corresponding impact to higher level *functionality* and *events* (to include kinetic or physical events), which we will explore in depth in Chapter 3.

A key challenge when dealing with uncertainty is retaining the ability to reason precisely. In our approach we specifically address three forms of uncertainty, mentioned earlier in this section:

Attacker To address *uncertainty in attacks*, we use a stochastic approach to represent attackers' points of presence as the probability a specific component is compromised. We also use a probability density function to represent the range of anticipated attacker capability (budget) that acts as a proxy for the attacker's ability to create novel exploits. This allows us to quantify our estimations of the attacker's locations and ability. At design time, this information would be collected from Subject Matter Experts (SMEs) as part of a threat assessment. At run time, updated information about the attacker is incorporated into the model of the attacker to refine the probabilities of her capability and points of presence.

Design Time System Uncertainty To address *design time uncertainty* in system implementation, our approach uses an architectural level of abstraction to abstract away implementation details that have not yet been determined. For example, we do not need to know that two components are running Microsoft SQL Server 2019 at a particular patch level. Instead, we simply represent that these are two devices of the same type, so we know they are susceptible to the same vulnerabilities. A vulnerability can be modeled based on its impact (e.g., confidentiality, integrity, availability) and cost to exploit. The cost of exploit is estimated by a subject matter expert based on open source intelligence, vulnerability history, attack surface, and other factors.

Run time System Uncertainty To address run time uncertainty in system implementation, we continue to use the same approach to vulnerabilities, using an architectural level of abstraction to enable reasoning about the relative hypothetical vulnerability of components and architectures without requiring knowledge of specific, known vulnerabilities. The model can be refined via humans in the loop or via automated means as new information becomes available. For example, if exploit code is published on the internet (so an attacker can download it without the cost associated with developing an exploit), the model can be updated to reflect that this exploit is available with no cost.

With this approach, we are able to formally represent the uncertainty relevant to graceful degradation for security. Further, we now have a way to represent our system with a consistent set of formal models in the form of architectural *views* that are interoperable with each other in a way that makes automated analysis possible for graceful degradation for security. These views allow for separation of concerns, partitioning of documentation effort among subject matter experts with differing expertise, and consistency with current practices for documenting systems.

We expand on the description of our approach in Section 1.2 and throughout the rest of this document. The remaining subsections of this chapter describe the problem and our approach in greater detail.

1.1 Motivating Example

In November 2011, as National Aeronautics and Space Administration (NASA) prepared to launch the Mars Science Laboratory (MSL) with the Curiosity rover, NASA's Jet Propulsion Laboratory (JPL) discovered it was under attack. According to the NASA Inspector General (IG) report, "...intruders had compromised the accounts of the most privileged JPL users, giving the intruders access to most of JPL's networks" [3]. NASA and JPL had to act quickly to identify where the attackers had presence, isolate possibly-compromised subsystems from mission-critical subsystems, and determine if the remaining trusted subsystems could safely accomplish the mission of launching MSL to Mars. The process of responding to attacks like the one at JPL is very human-intensive and not something that current risk management toolsets are customized to handle.

This incident shows how architects and administrators of critical systems struggle to prepare for and respond to the inevitability of attacks. In particular, two problems loom large – each motivating a set of research questions/challenges.

First, system architects need ways to evaluate and build graceful degradation capabilities into systems at design time so the systems are architected to be able to withstand attacks by gracefully trading off some system functionality to preserve other functionality. These capabilities would perform much like the crumple zone in a car – one section of the architecture (the engine block) is sacrificed to save another, far more important, part of the architecture (the human occupants). Current approaches do not enable this kind of thinking at Preliminary Design Review (PDR) phase,[1] in large part due to the high levels of uncertainty in implementation detail. These concerns motivate the following two Research Questions (RQs):

RQ 1: How to automate the evaluation of graceful degradation abilities in a system at PDR?

[1]"The purpose of the PDR is to demonstrate that the preliminary design meets all system requirements with acceptable risk and within the cost and schedule constraints, and that it establishes the basis for proceeding with detailed design. The PDR will show that the correct design options have been selected, that interfaces have been identified, and that verification methods have been described." [4] In other words, key architectural decisions have been made, but the detailed design / implementation is not yet complete.

RQ 2: How to automate discovery or selection of reusable graceful degradation architecture tactics?

Second, system administrators (defenders) need ways to evaluate the impact of attacks and deploy COAs in real time or near real time. As the JPL example demonstrates, there are many considerations that must be taken into account when defending against attacks – such as balancing risk with the goal of completing critical tasks – and this level of complexity can quickly overwhelm humans. These concerns motivate two additional research questions:

RQ 3: How to automate the incorporation of attack information in a way to provide defenders with actionable understanding of the impact of the threat to the system and missions?

RQ 4: How to automate the incorporation of attack information to generate real time COAs that maximize the system functionality while repelling or containing the attack?

We believe a formal, tool-supported approach to graceful degradation can address these four issues. Ideally, this approach would (1) utilize formal specifications of systems, (2) work effectively despite uncertainty regarding threat actors and threats, (3) provide useful architecture and design guidance at PDR, (4) provide for formal reasoning about attack implications in near real time, and (5) output effective COAs for defenders; these COAs maximize system functionality while maintaining an acceptable level of risk under attack.

Unfortunately, current approaches in both practice and research fail to realize these goals. As we elaborate in Section 2, current approaches to the evaluation of system security properties generally fall into one of two categories: top-down and bottom-up. Top-down approaches such as Fault Tree Analysis (FTA) and Systems Theoretic Process Analysis (STPA) are primarily intended for non-networked systems in which physical failure modes were the primary concern. They do not account for determined adversaries that target specific weaknesses and move through systems in a manner inconsistent with classical failure propagation. Additionally, these approaches were designed to be manual approaches that are implemented at the system design phase and revisited only as assumptions or implementation details change significantly. The information is captured in charts and spreadsheets by subject matter experts.

Meanwhile, bottom-up approaches such as Failure Mode, Effects, and Criticality Analysis (FMECA) and Event Tree Analysis (ETA) also do not model attackers and attack propagation; as manual approaches that could start with any possible fault, they do not scale to the needs of modern systems. Other bottom-up approaches that rely on the generation of attack scenarios assume detailed, knowledge of specific implementation details, specific vulnerabilities, and/or attacker TTPs. The information is recorded in a standard format such as those found in the Security Content Automation Protocol [5], and formal methods can often be used for evaluation of attacker movement through systems. However, these approaches do not incorporate the notions of system functionality and failure modes/events that are necessary to make informed trade-off decisions. Further, the amount of certainty in knowledge required is unrealistic at either the design or operational phases of the system lifecycle.

We bridge these two worlds by formal modeling, leveraging an interoperable set of architectural views ranging from enterprise concerns to system implementation details. Data flows and attack traces criss-cross views, enabling us to estimate the impact of various attack scenarios on the abilities of different architectural alternatives to trade off functional requirements with risk.

As we detail later, we have a proof of concept method and tool for modeling and evaluating system degradability at an architectural level of abstraction. This method addresses uncertainty regarding both the system implementation details and the nature of the adversary. Our approach is focused on realistic defense scenarios in which most or all vulnerabilities are hypothetical,[2] and each vulnerability has an estimated cost the attacker must "pay" to exploit it. Attackers have limited budgets and spend those on exploiting vulnerabilities. Our approach is generalizable and can be applied to different types of systems and throughout their life cycles.

1.2 Approach Overview

We argue that between the top-down and bottom-up – and the high- and low-level – approaches just outlined, there is a sweet spot within which we can use formal methods and an architectural level of abstraction – despite high levels of uncertainty – to formally represent systems' architectures in a way that enables evaluation for secure graceful degradation. An architectural level of abstraction hides the superfluous or unknown details of system implementation. To support integration and formal reasoning for graceful degradation for security, we implement our approach using Datalog, a declarative logic programming language that consists of rules (implications) applied to a database of facts.

The attacker's points of presence in the system are potentially uncertain. At design time, they are purely hypothetical targets for an attacker's points of entry. At run time, they are adjusted in real time based on the latest threat intelligence. These points of presence are the starting points for *attack traces*, in which the attacker uses her budget to craft exploits that enable her movement through the system. Because this budget is not fully known to the defender, we treat it as a probability density function, which the defender can adjust as he learns more about the attacker.[3]

Additionally, we assume defenders do not fully know the latent vulnerabilities in their systems; this assumption is reasonable for similar reasons at design and run times. We instead assume that each component has a vulnerability (or vulnerabilities of different types) with a

[2]We assume defenders will patch known vulnerabilities expeditiously.

[3]This is something that the defender would manually adjust in our approach.

cost associated with exploitation; that cost is borne by the attacker via the attacker's limited capability, which acts like a monetary budget, constraining the attacker.

The core of our approach is similar at both design time and run time. It ingests the above information and generates plausible attack traces – those that begin at possible points of attacker presence, grow through leveraging the hypothetical vulnerabilities, and are bounded by the expectation of the attacker's capability. We use Datalog, a declarative logic-based programming language, as a formal reasoning system to generate attack traces and determine the impact of each trace on the defended system's utility.

The attack traces impact data flows, compromising the defender's ability to deliver services that rely upon security attributes like confidentiality, integrity, and availability of the data they consume. The loss of the ability to guarantee security attributes leads to the compromise of system functionality, as subsystems can no longer be trusted to fulfill their functions according to specification. For each possible level of attacker capability, we assume the attacker follows the worst case scenario (to the defender) attack trace.

The utilities corresponding to the worst case attack traces are then weighted based on factors like the probability that the initial points of presence were compromised. The sum of these weighted utilities provides an estimated residual utility of the system – the utility that is expected to remain after an attack. We assess a variety of defender tactics to change the system architecture in response to the attack. These tactics include things like topological changes to system architecture – e.g., adding or removing communication channels (or connectors) between components. In many cases, an attacker many need to excise a compromised subset of components to bound risk while maximizing the estimated residual (remaining) utility of the system. Our approach determines appropriate ways for a defender to increase resilience of a system in anticipation of attack and respond to attacks through shedding functionality in a systematic and coordinated manner. These courses of action can either be generated and applied automatically, though a human operator could review the courses of action prior to application if there are concerns about automated courses of action.

1.3 Thesis Statement

We can architect and operate systems that are better able to weather attacks by automating the evaluation of systems' security properties to enable effective automated graceful degradation of systems in the presence of uncertainty through an approach of formally modeling systems and system behavior at an architectural level of abstraction to explore hypothetical attacks and the systems' abilities to respond.

Important properties of this approach include scalability and performance, realism, usability, and effectiveness.

1.4 Elaboration of Thesis Statement

By "architect and operate," we apply our techniques to both the design time and run time phases of the system lifecycle. An architect should anticipate that the system will come under attack, so the architect must ensure that the system has the appropriate capability to deflect and absorb the impact of a successful attack.

We use the following definition for graceful degradation:

> ...the term graceful degradation means that a system tolerates failures by reducing functionality or performance, rather than shutting down completely. In order for graceful degradation to be possible, the system must have some level of auxiliary functionality; i.e., it must be possible to define the system's state as "working" with other than complete functionality.[6]

With respect to graceful degradation in a security context, the defender must be careful to balance the risk of attack with the requirements to provide – and expose to attack – the functionality of a system. Further, what makes graceful degradation *graceful* is the *fine-grained* nature of the degradation. The impact of graceful degradation to the system utility should be the smallest amount that keeps risk within predetermined bounds. For example, rather than shutting down a system completely in response to an attack, it may be preferable to shut down only the compromised sub-systems.

Formal modeling of systems is broadly understood in the software and systems engineering communities, but formal modeling of system behaviors may be less familiar to software engineers. The formal modeling of behaviors links the contributions of subsystem components to the complex needs of entire systems. These contributions form a hierarchy. For example, a web service may require a front-end web server and either of two database servers. A company may require the web service (and therefore what that web service requires) and an email service.

Although there is often *uncertainty* in a contested environment (i.e., one without a clearly dominant player), we intend our uncertainty to match the kind that architects and administrators contend with in their day-to-day work. For example, system vulnerabilities and attacker TTPs may be unknown to the defender. Our approach reasons about this uncertainty through the generation of *hypothetical attacks*. Hypothetical attacks are those that are built upon a combination of hypothetical exploited vulnerabilities and attacker TTPs. These are instantiated in the form of *attack traces*, which are lists of hypothetical and/or known vulnerabilities that may be exploited by an attacker to achieve a known or unknown goal. Our approach ensures a level of realism, and it makes our work stand out from other approaches. A more in-depth discussion of how we approach uncertainty in representing system and attacker data can be found in Sections 3.7 and 3.10.

Scalability is the ability of our approach to reason about larger and/or more complex systems and provide output in a timely manner; *realism* is the ability of our approach to model attributes of systems and attackers and address the concerns of defenders in real world environments, *usability* is the ability of a defender to apply our approach to their system, and *effectiveness* is the ability of our approach to provide risk mitigation feedback that can meaningfully improve the system's security.

Scalability / Performance

Even relatively small systems can represent the types of security issues that vex architects and administrators to this day. Despite a small number of components, the number of possible connections is still 2^n, where n is the number of components in the system, so a naïve approach will not easily scale to a large number of components. However, we demonstrate that our approach produces meaningful results with small representations of complex systems.

Measure: This approach should work with systems large enough to meaningfully reason about the security implications of various key architectural decisions. Detailed, side-by-side comparisons of architectures (carried out at design time) should be computed within minutes or hours in the compute environment provided by a small cloud network (e.g., five medium server instances). COA recommendations should be output within minutes.

Realism

This methodology and tooling should be applicable to critical cyber-physical systems in multiple domains. It should be able to address the concerns associated with deployed systems and those intended for deployment in production environments.

Measure: The methodology and tooling should be generalizable to different types of systems. We will demonstrate success through the application to three domains: aerospace, electrical utility, and military.

Usability

The methodology and tool should integrate with and leverage existing processes, procedures, and tooling on small-scale systems – like those used in training and simulation environments – as well as smaller deployed systems. We use the term usability to mean practicability; while we do not evaluate with users, a successful approach tries to minimize the burden for new sources of inputs and requirements for additional labor.

The approach should be something that can be applied with minimal subject matter expertise. The need for some expertise is inescapable, but the demand should not be onerous. Further, it should integrate with and leverage existing tooling and processes.

Measures: Systems are described such that aspects that require SME input are separated from those that do not. SME input should be required once to initiate the modeling effort; from that point on, non-SME personnel can use the tooling. Labor should be minimized through the ability to reuse SME-generated inputs and – with future integrations – automate of other inputs. These measures can be evaluated with case studies that prove the concept of the approach.

Effective

The methodology and tool should provide outputs that aid the defender in anticipating and responding to attacks through the output of COAs that protect higher utility functionality by trading off insecure or questionably secure functionality.

Measure:

COAs provided by the tool align with best practices for secure system architectures.

1.5 Contributions

The contributions of this research are:

- We propose a small subset of information that must be represented to effectively reason about graceful degradation for security. This knowledge explicitly incorporates uncertainty that reflects the realism system defenders face today. Much of this information is formally represented via architectural views.
- We provide formal representations of this knowledge at an architectural level of abstraction in the form of architectural views. The views are interoperable and integrate information of disparate types including deployment/allocation, system functions, data flows, and events.
- We describe a method to evaluate the ability of a system to anticipate and respond to attacks – despite uncertainty – by a coordinated trade-off of functionality to manage risk from attack.
- We demonstrate and evaluate the proof of concept of our method via a tool we developed.

In the remainder of this document we describe related work in Chapter 2, outline our approach in Chapter 3, explain how we implemented our approach in Chapter 4, validate our approach in Chapter 5, and close with additional discussion of our work and opportunities for future work in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we will elaborate on the motivating example to gain understanding of the root problems with graceful degradation in security. We explain why this is a problem worth solving, and we argue that prior research addresses aspects of – but does fully solve – our motivating problem. We then describe a set of high level requirements for a better solution that addresses head-on the shortcomings identified earlier.

2.1 Motivating Example Discussion

In our motivating example in Section 1.1, we described an attack at National Aeronautics and Space Administration (NASA) Jet Propulsion Laboratory (JPL). This attack occurred at a crucial time for a launch to Mars, leading to high-stakes decision-making with time pressure. The introduction of a small delay in the launch date to allow time for investigation and securing all systems could have pushed the launch out a year and a half. On the other hand, a rushed and insecure launch could have – in a worst case scenario – led to the loss of a multi-billion dollar mission. In the end, JPL made a decision to partition its network, keeping the attacker away from critical systems.

However, network partitioning can result in the loss of some functionality, and this must be implemented cautiously to avoid partitioning critical systems from each other or allowing the attacker a foothold in a critical area of the system. Automation might have helped to assess the situation and implement a response faster and with more assurance that the result would be a successful launch with minimal degradation to JPL systems. This is not an isolated problem. In recent years, attacks of large critical systems have become commonplace (e.g., Stuxnet (ICS), Sandworm, the Target credit card breach, infiltrations of Ukrainian power plants, and an attack on the Viasat satellite communications system).

To be able to address situations like these effectively in the future, a successful approach must advance beyond the state of current practice and fulfill a number of requirements, which we describe in the following subsection.

2.2 Requirements

First, we need a way to automate the analysis of the security properties of architectures and the means of graceful degradation of those architectures in response to attacks. Systems are already very complex, and humans may not have the time to assess the effects and side effects of graceful degradations they manually implement. Additionally, unlike manual interventions, automation – if done right – should also be repeatable, explainable to defenders, and defensible (i.e., reasonable given the input circumstances).

Second, a successful approach will integrate information from a variety of sources to provide effective degradation strategies. This includes information about how an attacker might move through a system, how a defender might change a system to respond to an attack, how system functionality changes as the system is attacked, and how system functionality changes as the defender changes the system.

At design time, organizations like JPL have processes like Preliminary Design Review (PDR) to evaluate high level architectures of systems for readiness for further design and subsequent implementation. This is a logical point in time to evaluate an architecture for its ability to withstand and gracefully degrade in response to attacks. To successfully automate this, an approach should leverage the artifacts that are already produced at this point in time. Further, the best approaches work well even with Agile development processes by not requiring significant additional labor or interfering with the Agile process.

We assume that automated graceful degradation for security will be used by organizations designing and operating systems that 1) are of a scale complex enough to warrant a thoughtful architecture, and 2) are critical so that the degradation or failure of the systems could lead to serious injury, large scale financial impact, or loss of life. These organizations should be using "architecture-focused design" principles [7].

At run time, the approach updates its model of the system and environment based on new information, such as updated threat or attack information. It generates effective Courses of Action (COAs) that can 1) adapt the system to become more resilient against the updated attacker or 2) mitigate ongoing attacks through a graceful degradation of system functionality. These COAs must be available within a matter of hours or less to effectively defend against modern threats [8]. To convince skeptical system administrators, evidence is produced in a way that can explain COA reasoning to a human.

Regardless of the software development life-cycle process, handling uncertainty is key to success. According to one report produced for NASA, PDR "is conducted when the design layouts are 95% completed and the detailed design is 10% completed" [9]. In an Agile world, a "risk-driven model of architecture guides developers to do just enough architecture then resume coding" [7]. Thus, architectural documentation will be – at best – light on implementation details. Successful approaches are able to produce effective guidance based on information that modern-day architects and system administrators have available in the real world. This includes treating unknown vulnerabilities as first class and being able to evaluate despite uncertainty around attacker Tactics, Techniques, and Procedures (TTPs).

To properly model security, successful approaches model specific attributes. "Before proceeding with discussing what to model, it is important to distinguish between prevention and detection of a transition into an insecure state" [10]. This corresponds to our requirements that a successful approach can work at both design time (prevention) and run time (detection, though our focus is mitigation) – with the ability to adapt at run time as new information emerges. Model requirements are broken down into the categories of target system, users, adversaries, and measures/countermeasures [10].

Finally, to be able to gracefully degrade, models must contain the necessary attributes to ensure evaluations do not simply result in all-or-no functionality. Successful approaches model nuanced tradeoffs so a system can continue to provide a subset of functionality securely by trading off a different subset of functionality. We expect these to align with common degradation measures like perimeters, tiers, redundancy, diversity, etc.

2.3 Prior Work

The studies of resilience to system failures and graceful degradation have existed for years, and our research builds on these concepts. Resilience has been an area of study since systems became complex enough to warrant understanding how failures propagate as faults through systems until bad things – known as "losses" – occur. By the mid 20th century, techniques had been developed for predicting the resilience of complex systems to catastrophes caused by natural phenomena. Subsequent work addressed graceful degradation in the physical world. Often, these forms of analyses are high-level tabletop-style exercises involving high degrees of uncertainty. As cybersecurity became an issue, techniques were developed to formally evaluate systems, as well as predict attacker movements. One approach, the use of attack graphs, is applicable directly to cybersecurity; it formally models systems and assumes a high degree of system implementation knowledge and certainty; it can automate understanding low-level impacts of specific attacks [11].

To successfully automate secure graceful degradation, we need a formal approach that requires only the realistic amount of knowledge and certainty that defenders can be expected to have, while limiting the labor and compute resources required to use the approach. For effective evaluation of trade offs of functionality for security, the impacts should tie back to high-level system requirements.

2.3.1 Resilience

Since the dawn of the Cold War, a number of techniques have emerged to evaluate and mitigate the causes and consequences of faults. These techniques were developed and honed for complex systems of systems like rocketry. These techniques fall into two basic categories that each answer a different question.

Bottom-up approaches, sometimes referred to as *inductive* approaches, begin with some *failure* or *initial event*, and they move forward in time, evaluating the cascade of *faults* or *pivotal events* that lead to a *hazardous state*, in which a *loss* (e.g., of life or system components) is possible [12]. These techniques address the question of what happens if a failure occurs.

A well-known inductive approach is a Failure Mode and Effects Analysis (FMEA) or Failure Mode, Effects, and Criticality Analysis (FMECA) when the criticality of the effect is considered [12]. These types of techniques usually begin with the identification of a single failure at a time, describing the modes (ways) the component might fail, the probability of component failure for each mode, and the category of the resulting effect (e.g., critical or not critical). This process is entirely manual.

Top-down approaches, sometimes referred to as *deductive* approaches, go chronologically backward by starting with identifying the possible losses and related hazards, and then recursively answering the question of *What could cause this?*, continuing backward in time until an initiating event or failure is identified. These techniques address the question of how a particular hazardous condition might occur. Fault Tree Analysis (FTA) is a well-known example of an inductive technique [12].

> The fault tree itself is a graphic model of the various parallel and sequential combinations of faults that will result in the occurrence of the predefined undesired event....
>
> A fault tree is a complex of entities known as "gates" which serve to permit or inhibit the passage of fault logic up the tree. The gates show the relationships of events needed for the occurrence of a "higher" event. The "higher" event is the "output" of the gate; the "lower" events are the "inputs" to the gate. The gate symbol denotes the type of relationship of the input events required for the output event. Thus, gates are somewhat analogous to switches in an electrical circuit or two valves in a piping layout. [12]

Another top-down approach gaining popularity is Systems Theoretic Process Analysis (STPA), a qualitative technique [13]. Like other deductive techniques, STPA begins by defining the losses and hazardous conditions that must be avoided. Rather than focusing on component failures, STPA uses a control system model; this system control structure – including humans in the loop – is recursively refined to the depth necessary for analysis. *Unsafe Control Actions* are identified, and these are used to create *Accident Causal Scenarios*. Systems Theoretic Process Analysis-Security (STPA-Sec) is a security-specific variant of STPA [14]. STPA-Sec is being used by the US Air Force as a key aspect of its Mission-based Risk Assessment Process for Cyber (MRAP-C) risk assessments [15].

Some approaches, such as Probabilistic Risk Assessment (PRA), combine bottom-up and top-down approaches [16]. PRA begins with the construction of a Master Logic Diagram, which is used to work deductively from undesirable end states through system functionality and subsystem hierarchy until Initiating Events are identified. Then, event sequence diagrams or event trees inductively identify the Pivotal Events between each Initiating Event and a corresponding series of possible end states. For each Pivotal Event, a fault tree deductively determines the system components that must fail – and their failure modes – to cause the Pivotal Event.

All of the above approaches are highly manual, requiring meetings of Subject Matter Experts (SMEs) each time an analysis is conducted. They are generally performed at the architecture and design phase of the system life cycle, though they could be used to evaluate deployed systems. Each re-evaluation of a system after a modification results in the need for SMEs to meet once again and determine the impact of the system or environment modification. These approaches are not automatable or intended to generate results to deny or deter attacks. With the exception of STPA-Sec, they are not security-specific, so there is no modeling of an attacker.

2.3.2 Degradation

Degradation has been studied in non-security contexts, and there are useful lessons-learned from the prior work. The Simplex Architecture aims to provide a means for safely upgrading a system while it is running [17]. In Simplex, if the experimental (new) controller is problematic, the system falls back to a safety (backup) controller while the experimental controller is fixed. This can be an all-or-nothing ordeal for the functionality provided by that controller, and the higher-level impacts of the fall-back are not a part of any evaluation.

Subsequent research specific to graceful degradation focused on how to scale it. This work focused on physical systems. A key insight was to leverage an architectural level of abstraction to hide subsystem complexity where possible, enabling analyses of graceful degradation properties to scale to larger systems of systems such as automobiles [6].

Graceful degradation is different in the context of security, and the prior work here does not sufficiently address this for our needs. Most resilience and degradation approaches are intended for use in situations in which faults are the result of stochastic processes. However, attackers may deliberately avoid doing what is expected for fear of discovery. This means that attacks do not follow nice Gaussian distributions that are easily predicted and modeled using traditional means.

Additionally, the propagation of failures in traditional contexts is different from that in a security context. Attackers move in ways that are constrained by a system's physical *and* virtual topology, with impacts that also propagate through a hierarchy of system functionality. Since exploits are reusable with little to no marginal cost, faults of similar components have little independence. As with resilience techniques, degradation techniques do not meet the requirements for a success in a security context.

2.3.3 Formal Verification

Application-Specific Verification

One way to approach security is through formal verification of that system specifications guarantee specific properties or that a system implementation meets the specification requirements. These have historically been difficult undertakings for even small or moderate sized software projects. For example, the verification of the seL4 microkernel took about 20 person-years [18]. This type of approach is not feasible for an organization performing a PDR-style analysis, much less one trying to make run time decisions on how to gracefully degrade.

To reduce the labor required for small applications, the developers of Ironclad Apps took a different approach [19]. As part of the application development process, they first created the specifications and code in Dafny, a high-level language with built-in verification by the Z3 Satisfiability Modulo Theories (SMT) solver. An untrusted compiler then builds out the code in a verifiable manner, and this code goes through a verification process to ensure it meets the specifications. The developers of the Ironclad Apps approach claim significant savings in time for the development of verified applications. Because the design is not specified down to the code level at design time – and we assume a level of implementation uncertainty at run time – the technological limit of formal verification of large scale systems is likely to be a verification of specifications. Regardless, this approach requires considerable time and advanced knowledge to implement.

This has two bearings on our approach. First, most software is currently not formally verified to this level, so there are likely to be many latent vulnerabilities remaining in systems to be defended. Second, the Ironclad approach demonstrates the utility of using a model of the system – in this case, in the form of the specifications – to understand and constrain system behavior, as well as to identify problems in a manner that is less demanding of SME labor. However, a number of properties can be formally verified, so formal verification does not necessarily imply graceful degradation unless that is a property to be specifically verified.

These approaches require high levels of expertise and significant amounts of labor from SMEs, and they are not meant for use in complex, highly uncertain environments.

Vulnerability and Attack Surface Analysis

Prior work has explored using attack graphs to evaluate the impact of known vulnerabilities within systems [20]. Attack graphs describe the application of exploits to vulnerabilities and subsequent movement of attackers through systems. This work has focused on how to describe vulnerabilities and system architectures, but not been used extensively for attempts to mitigate vulnerabilities architecturally through graceful degradation.

MulVal [11] builds on the concept of attack graphs; it takes as input a description of the system architecture and vulnerability information along with security policy requirements; it either finds no policy violations or it outputs attack counterexamples. MulVal includes a capability for reasoning about hypothetical vulnerabilities for predicting the impact of a particular hypothetical vulnerability (or vulnerabilities) on a system.

MulVal shares some goals and design choices with our approach — in particular its use of Datalog as both a specification language and reasoning engine. However, we differ significantly in several ways. We introduce "functionality" as way of understanding the relationship between system connectivity, component compromise, and utility; this enables evaluations of scenarios like the check clearing scenario proposed in Jha [21], in which connectivity is key to evaluating system utility. We also make the secure flow of data central to our evaluations of system utility. Additionally, our approach is geared toward finding an architectural alternative that gracefully degrades the system by shedding some functionality during compromise to maximize remaining utility.

Another thread of research explores how defenders can evaluate the security resilience of architectures with respect to zero day vulnerabilities. The k-Zero Days approach uses the number of zero days an attacker would need to move from one point in a system to another as a metric for the resilience of that system to attack [22]. We leverage a similar insight to evaluate graceful degradation, which transitions a system from one architecture to another.

These approaches do not make uncertainty first class elements, and they do not have a capability to demonstrate the concept of graceful degradation through the application of COAs.

Other related research focuses on how to evaluate the impact of vulnerabilities on the ability of a system to execute its functionality in an assured manner [21]. This research uses computation tree logic (CTL) to describe guarantees that a system must uphold. These guarantees include nuances like the connectivity between nodes. While this research does not present a complete theory of how to describe functionality, nor does it provide a method for adapting architectures to become more resilient, it provides a step forward through the use of sophisticated means of describing the types of guarantees a system must uphold to provide its full utility.

Other research has focused more on adaptation rather than static resilience. One approach is the Simplex architecture [23]. Simplex provides a fallback mode that can be switched to in real time. It could be considered to be a particularly coarse method of graceful degradation [17]. This approach is akin to the use of a safe mode, albeit with additional complexity.

The Rainbow framework leverages a model of the architecture of a system to predict future state and ensure appropriate adaptive reactions based on the current state [2][24]. This is a general-purpose approach focused only on run time and not specifically geared toward modeling attacks to gracefully degrade system functionality.

Robustness

Another promising approach based on formal verification is behavioral notion of robustness [25]. This approach formally defines robustness as “the largest set of deviating environmental behaviors under which the system is capable of guaranteeing a desired property.” Robustness can be used to understand how a system will respond to specific environmental triggers and to ensure safety properties given those triggers. It is not a security-specific approach, so it is not suited for evaluating how attackers may move through systems. While robustness and graceful degradation are related concepts, this approach does not incorporate notions of utility or fine-grained functionality for the purpose of evaluating how to gracefully degrade systems under attack.

2.4 Gaps and Remaining Problems

While each of these works represents an important contribution, a number of problems remain unsolved.

A variety of the high-level (architectural level of abstraction) approaches leverage human experts in manual processes to evaluate snapshot-in-time systems in ways that explore graceful and catastrophic degradation [26][27][28][16][14]. Furthermore, most of these approaches focus on natural phenomena that can negatively impact systems. Natural phenomena occur in ways that lend themselves well to probabilistic analyses (e.g., mean time to failure of a component), but attack analysis needs to consider worst-case, not average-case outcomes.

Additionally, these approaches are not designed to be automated. They require experts to not only describe the architectures and probability distributions of various phenomena, but – in many cases – to also walk through the analyses step-by-step to understand the follow-on impacts or contributing factors to a particular step.

The more automated approaches like Mulval do not treat uncertainty as first class and are also unable to model graceful degradation [11]. Of the more quantitative approaches that are generally designed for use at run time, many assume knowledge of system implementation details, including the knowledge of system vulnerabilities. For these reasons, the approaches are not appropriate at design time, and the level of expected knowledge is unrealistic at run time. Further, many of these quantitative approaches do not factor in the critical functional and mission knowledge necessary to enable graceful degradation for security.

2.5 A Path Forward

We believe automation of self-adaptive secure graceful degradation is possible by leveraging formal reasoning to integrate the various advances in the prior work. The solution lies between manual top-down approaches that incorporate attributes like functionality and bottom-up approaches that enable tracing attacks through deployed or to-be-deployed system architectures. A successful approach incorporates the knowledge and uncertainties captured in enterprise architecture levels with knowledge and uncertainties captured in system architectures, leverages existing processes and tools, is security-specific, is highly automatable, and can be applied at both architecture/design and run time. Further, a successful approach can complement a human-in-the-loop approach, but it can also be run with a human out of the loop.

Chapter 3

Approach

In this chapter, we explain our approach and provide the rationale for key decisions we made. The following chapter, Chapter 4, contains an in-depth technical discussion. We begin the current chapter by briefly describing the primary goals of our approach. Next, we provide usage scenarios to describe the approach's intended use cases. This is followed by the key principles and design constraints we place on ourselves. With these in place, we describe the high level approach we took. Later, we introduce an example system to be defended and expand on some of the most important design decisions we made on how we model systems and their environments, using the example for illustration. Finally, we introduce the proof-of-concept tool we developed to implement our approach.

3.1 Approach Goals

Our primary goal is for our approach to provide effective guidance for software architects and system administrators who must understand and respond to the impacts of attacks by gracefully degrading system functionality. For systems in the architecture and design phase, our approach should identify ways to build graceful degradation for security into the system architecture, identify and mitigate points where security resilience can be increased, and understand the mission impact of different attacker scenarios (i.e., attacker starting points and capabilities) before the system is implemented. For operational systems, our approach should help defenders understand the impacts of attacks on system operations and develop one or more Automated Courses of Action (ACOAs) to deny, deter, or delay the attacks.

3.2 Usage Scenarios

The usage scenarios in this section provide an overview of how our approach would be used in a real-world environment. We describe how our approach's pieces fit together into a self-adaptive system, what the inputs and outputs are of that system, and how users would interact with that system.

A high-level diagram showing the inputs, flow, and processing of information, and outputs is shown in Figure 3.4 in Section 3.11.

3.2.1 Architecture and Design Time

The goal of the architecture and design time use of our approach is to guide architects and designers to create resilient and gracefully degradable systems. This is achieved by evaluating and comparing how different architecture alternatives resist attacks.

At design time, the primary stakeholder is the system architect and/or engineer. They may begin with no predetermined final architecture, and their understanding of the attacker (e.g., which component is the attacker's target, is the goal destruction or intelligence gathering, how capable is the attacker) is likely to be highly limited. Specific implementation decisions such as commercial / open source software brands and versions and subsystem architectures have yet to be made. However, architects are likely to have one or more options for comparison. These options form the initial input models for our approach when used in this phase of the system life cycle.

To evaluate the secure graceful degradation properties of the initial system under consideration, the architect must be able to model the salient aspects of the system. We go into more depth on this later in this chapter. Additionally, the architect must be able to document the consequences of the compromise of parts of the system. For example, the defender will need to understand if an attack's effects can be contained to a single subsystem or if the attack inevitably causes a catastrophic affect to the entire system. This can be determined without detailed implementation decisions.

Our approach evaluates the system model (including other relevant information about the attacker and defender), and it compares that with similar models generated to be within a specified graph edit distance of the original model. By evaluating how an attacker might exploit the system, the approach outputs system models that are adaptations of the input model, but these incorporate the attacks and may degrade the system by trading off some aspects of the system to protect other parts.

This approach is meant to be used as part of an iterative process of specification and generation. As implementation decisions are made, these decisions are reflected in manual updates to the model representing the defended system in our approach, resulting in improved guidance regarding security resilience and graceful degradation. For example, if the architect learns which components use the same software (e.g., the same operating system or service), this information can be used to identify common points of failure for which an adversary could reuse a single exploit multiple times.

3.2.2 Run Time

The goal of the run-time use case of our approach is to guide or augment system administrators in responding to ongoing attacks once a system is deployed. This is accomplished through using our approach to determine degradation options and compare the associated models' abilities to continue to fulfill system requirements. The best option found in the evaluation can be carried out with or without a human in the loop.

At run time, the system administrator is our defender. They must react quickly and are likely to be constrained by previous architecture and implementation decisions such as the operating system vendor of host components and current network topology. Unlike the prior work described in Chapter 2, our approach is meant to be effective even if most of the existing vulnerabilities are unknown.

The system administrator uses our approach to react to new information about the attacker. The sequence of events is shown in Figure 3.4 in Section 3.11. First, our approach begins with the model that represents the current architecture and its environment.

With these inputs, our approach generates a set of Courses of Action (COAs), which are sequences of tactics to change the system model to alternative architectures. Our approach evaluates the expected value of the system's utility – after factoring in possible anticipated attacks – of each of the alternatives. The COA with the highest estimated residual utility is the one output by the tool as the preferred COA for the defender to take.

As time is typically of the essence in responding to attacks, COAs or ACOAs can be generated in advance by simulating various specific scenarios that are considered the most likely to be encountered. For example, if Virtual Private Network (VPN)-based attacks are considered likely, and if the best response to an attack on the VPN is to remove the VPN service from the network, that ACOA could be generated and stored in advance – and simply looked up and applied in the case of an observed successful attack on the VPN.

3.3 Key Assertions

Degradation implies trade-offs. In a security context, our goal is to intentionally trade off insecure functionality to maximize remaining secure functionality within a predetermined level of certainty or risk tolerance. Our approach to automated secure graceful degradation explores the trade space between system functionality and security risk, and is based on the following key assertions:

Information requirements do not exceed what can be expected of today's defenders. We are cognizant that defenders have limited knowledge of attackers and even of their own systems. Therefore, we are careful not to require that the defenders produce information that they cannot be expected to know. There may be implicit information that a defender must make explicit

– like the expected capability of an attacker or the vulnerability of a component. However, defenders must already have rough estimates as part of their mental models today.

Uncertainty must be treated as first class. To be realistic, we directly incorporate uncertainty into our approach. Uncertainty is key to how modern systems are defended. After all, if a defender knows exactly how an attacker would attack, the defender will mitigate those vulnerabilities and nothing more. Similarly, if a defender knows specific vulnerabilities in their system, they are likely to mitigate those vulnerabilities. That leaves the unknown vulnerabilities and unknown attacks as key to what defenders must protect their systems from. Of course, this is not something typically quantified.

We represent vulnerabilities by the estimated "cost" to exploit a component in a particular way. This is akin to the cost to develop, buy, or otherwise obtain an exploit for that component. This may be estimated based on information such as attack surface, vulnerability history, and even exploit market value information [29].

Attackers are represented by their capabilities (i.e., budgets) and probabilities of having an established point of presence on specific components.[1] The points of presence represent anticipated starting points for any attacks. The uncertainty in capabilities is explicitly represented with a probability density function. This is because an attacker does not know the specific capability of the attacker, and there are possibilities of "Black Swan" (i.e., low probability, high impact) events that should be represented to ensure realism in the model. Uncertainty in attacker points of presence is represented by the percentage values assigned to specific components. These values represent the likelihood that the component is compromised at the outset of the analysis.

Architectural level of abstraction is the appropriate level of abstraction for secure graceful degradation analysis. An architectural level of abstraction includes "gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives." [2] We use an architectural level of abstraction in our models because it represents the things we care about (i.e., components, connections, data, functionality) while allowing us to abstract or hide information that either is not relevant to or unnecessarily bogs down evaluations of system alternatives. Furthermore, we expect this high-level architecture information is available at both an early stage of design time and during run time. Architectural levels of abstraction also hide superfluous details, including details about which there are uncertainties not directly relevant to our analysis. An example is software code; while vulnerabilities can arise from oversights in software, the uncertainty in whether and where vulnerabilities exist means we should not

[1]A capability is like a budget. An attacker can use their capability to create various combinations of reusable exploits as long as they stay within their budget.

focus on specific vulnerabilities – rather, we should focus on the extent to which components are vulnerable and how difficult it is likely to be to exploit those vulnerabilities.

Formalism is key to automation. To automate evaluations of graceful and secure degradation, inputs must be machine-readable. Formal modeling of the system and environment using architectural views – and definition of the attacker and defender profile using the parameters described above – results in quantitative, machine-readable formatting.

Attack scenarios are critical for evaluating security implications of architectures. Our approach constructs attack scenarios based on a small number of assumptions of how attackers and defenders affect system functionality. These attack scenarios are the core of how system architectures are evaluated for their security properties. A system with a number of plausible, devastating attack scenarios is less secure than a system of similar functionality and few plausible, attack scenarios of minor impact. Approaches to evaluating reliability use fault propagation through physical systems as their core; we use attack scenarios in lieu of fault propagation, resulting in an evaluation of security rather than reliability.

The integration of information from multiple views informs evaluation of secure graceful degradation. Secure graceful degradation implies trade-offs. Any sort of graceful degradation must be based on a trade-off that optimizes functionality given security realities that reduce functionality. In our attack scenarios, attacker movement through a system can be clearly represented in an allocation view. System functionality is represented in a separate, but related, functional view. Data flows to and from components are represented in component-and-connector views. *Functionality vis-à-vis the data flows is what is ultimately being protected – not a specific component.* Components are integral to functionality because of the data flows they produce, consume, or transmit. An attack that compromises a particular component also compromises the data flows to, from, or through that component; this compromise of a data flow subsequently compromises functionality. Thus, there is a relationship between the allocation views, functional views, and component-and-connector views. Therefore, our approach must integrate these architecture views to evaluate and compare the security and functional implications of architecture alternatives.

Simple assumptions regarding system security explain best practices. Our approach makes a set of general assumptions about the elements that a model should contain and how those elements interact. These are similar to axiomatic principles, and they are not domain-specific. We will show how it is possible to automate secure graceful degradation – and provide explanation-aiding artifacts – using analyses of attack traces across an integrated multi-view architecture. By considering the impacts of possible attacks on system functionality, we perform side-by-side comparisons of different architecture alternatives to determine the best one to degrade to based on the available information.

In the following sections, we describe our approach in more detail, showing how our tool implements our approach.

3.4 High Level Approach

In this section, we provide an overview of our approach that will be elaborated on further in the rest of this chapter. Along with our approach, we provide a tool that ingests and integrates the system model in the form of architecture views, styles, and associated data to make graceful degradation decisions. An architecture view is "a representation of a set of system elements and the relationships associated with them," and a style is "a specialization of element and relation types, together with a set of constraints on how they can be used" [30]. An example of a style is a client-server style; clients are constrained to connecting to servers, whereas servers can cannect with clients or other servers. Styles are associated with views and place constraints on architectures; they generally stay static during a system lifecycle while views may change. Architectural views and styles are standardized in ISO/IEC/IEEE 42020 [31]. The architecture views and styles are currently manually created by architecture and system administration Subject Matter Experts (SMEs). The parameters that describe the attacker are input to the interpreter that runs the tool's analysis.

Given an initial architecture and a graph edit distance parameter to constrain permutations from the original architecture, the tool generates architecture alternatives to which the defender may wish to degrade. At architecture and design time, this distance could represent the number of architectural permutations (e.g., adding and removing connections) from the initial architecture. At run time, this could represent the maximum number of architecture changes a defender make before the attacker makes other moves in the defended system. Different tactics may be assigned different costs based on their relative difficulty. For architecture alternatives, we currently consider different topological arrangements of components (i.e., adding or removing connections) but not changes to components themselves. This process generates and evaluates the estimated utility of all possible COAs constrained by the defender's budget for changes to the existing architecture and other user-definable constraints. This is currently an anytime algorithm [32], so the evaluation could be stopped at any time, and the best alternative found so far could be selected as the target of graceful degradation.

Our approach evaluates the initial and alternative architectures for their utility in the face of the anticipated attack scenarios. An attack scenario includes the attacker's possible starting points and series of movements and exploits (including stealing and reusing credentials) as they proceed to compromise more and more of the system. Attack scenario artifacts can aid in explanation and act as evidence for architecture and design milestone decisions (e.g., Preliminary Design Review (PDR)). The scenarios provide specific cases of attack – with documented impacts on system functionality – to be used by architects, designers, and system administrators as they degrade systems in response to attacks.

The best architecture is the one with the highest expected utility in the face of anticipated attacks. This is output as a COA with the associated resulting architecture, which is the one to which the defender should gracefully degrade.

When run at architecture and design time, the architect may use several runs of our approach using as starting points models that correspond to the architecture alternatives under consideration. Outputs can demonstrate which model is more resilient and how the systems should be adapted to be more resilient for the purposes of secure graceful degradation.

At run time, a system administrator will provide an initial input model that corresponds to the current system state. As new information (e.g., about the attacker) arrives or the system state changes (e.g., a server is shut down), the administrator will update the model and rerun the analysis. The output will show the administrator an option for degradation based on the inputs provided.

Inputs to our approach are as follows:

- An initial architecture, including expected locations of and usage of credentials, plus expected levels of difficulty for crafting exploits.
  - In the design time use case, this initial architecture is an architecture under consideration.
  - In the run-time use case, it is the current system architecture.
- An attacker profile, which consists of the expected attacker capabilities and points of presence is developed by a SME.
- A defender profile, which is a budget the defender has to implement defensive actions in response to an attacker.

With these inputs, our approach is outlined as follows:

1. From the initial architecture, generate a set of alternative architectures (described in Section 5.1.2) to consider for degradation.
2. For each architecture alternative, including the initial architecture:
   (a) Construct all possible attack scenarios given the attacker profile and the set of hypothetical system vulnerabilities.
   (b) For each attack scenario on a given architecture alternative:
      i. Determine the residual utility of the attack scenario on the architecture alternative.
      ii. Multiply the likelihood of the attack scenario by the residual utility.

   (c) Calculate the sum of the multiples calculated above. This is the estimated residual utility for the architecture alternative.

3. Output the architecture alternative and corresponding COA with the highest estimated residual utility. Provide evidentiary artifacts such as possible attack scenarios to aid in explainability of COAs and increase users' trust in system outputs.

There are key differences between the design time and run time modes of operation of our approach. At design time, the uncertainty is much higher for implementation details, specific vulnerabilities, and attacker profiles. There is also no need for a swift reaction to an attack yet. Because the system has not been built, the cost to change from one architecture to another is far less; it is a design effort rather than an implementation one.

At run time, the implementation is complete. Specific vulnerabilities are still unknown because known ones have hopefully already been patched. However, a security SME can estimate a component's vulnerability based on the attack surface and other implementation details. When an attack is underway, specific points of compromise become known with certainty, and the attacker's Tactics, Techniques, and Procedures (TTPs) can aid with the identification and estimation of the attacker's profile. While this decrease in uncertainty is beneficial, reaction time and the cost of a response are important factors in the defender's success in mitigating attack damage. Our approach must ingest the updated parameters and provide effective COAs in time to stop attacks. At the same time, COAs must be reasonable to implement; i.e., it would be impractical to provide a COA that is a ground-up restructuring of a system that is already deployed and operational.

3.5 Running Example

As a running example, consider the Industrial Control System (ICS) depicted in Figure 3.1. This system is representative, but simplified, of real grids in use today to control electricity transmission [33]. The system consists of remote components like relays that open and close to transmit or cut the flow of electricity to transformers (used to change the voltage) and Remote Terminal Units (RTUs) that monitor the system and environmental parameters. The Supervisory Control And Data Acquisition (SCADA) server provides moment-to-moment tuning of the control system, while a Human Machine Interface (HMI) exists for a human in the loop to monitor and adjust day-to-day operations. The Open Platform Communications (OPC) service translates data between the disparate formats produced and consumed by the ICS that is the overall electrical transmission system [33]. These architectural elements serve together to ensure proper electrical transmission functionality [34]. Further, an engineering workstation allows an engineer to adjust the parameters used by the SCADA server and make more changes to the ICS operations than the HMI, providing additional grid management functionality.

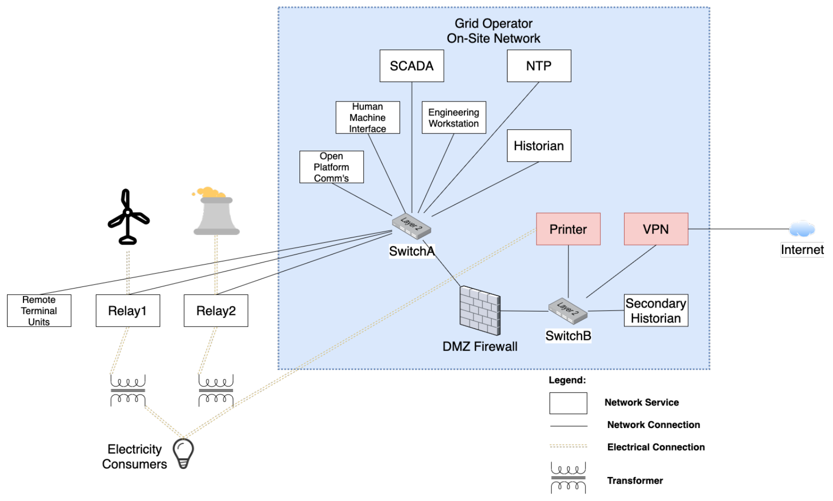

Figure 3.1: Deployment View of the running example ICS system.

Because electrical transmission is a business, there are additional enterprise functions. Examples of typical business functions include billing, accounts payable, payroll, etc., which keep the enterprise running. The printer is used for routine enterprise use and is not necessary to a functioning grid. The VPN service allows users to remotely access network resources. The historian is a type of database used for logging.

3.6 Rationale for Architectural Level of Abstraction

There are four primary reasons for why architectural models are appropriate for the problem we are trying to solve. First, as we claimed in Section 3.3, an architectural level of abstraction contains nearly all the data we require for our evaluations. Second, it contains limited superfluous data. Third, architectures are expressed in *views*, and these views are common to industry such as in [31], so they are well supported by other tools and are often created as part of the system life cycle. Fourth, these models are available and definable at the architecture and design phase and at the run time phase of the system life cycle.

Architecture and design models need only be refined to the point where they answer risk-based questions for graceful degradation. “You want to build successful systems by taking a path that spends your time most effectively. That means addressing risks by applying architecture and design techniques but only when they are motivated by risks” [7]. That concept to guide abstraction allows ambiguity (uncertainty) for some implementation details and reduces the work to apply our approach. Information requirements and estimated collection burden are described in Section 3.11.

We wish to balance the needs for usability and performance with the need to produce realistic and effective COAs. We balance these needs by carefully selecting the elements we model. In particular, as we noted in our requirements in Section 2.2, we must model the technical aspects of the system being defended (i.e., the system), the mission-oriented and functional aspects of this system, the users of this system, the attackers (i.e., adversaries), and measures/countermeasures. These break down into the following, adapted from [10].

- Target system: Data at rest (i.e., store), in transit (i.e., network), and in transition (i.e., compute/process). Functionality.
- Users: Credentials
- Attackers: Capabilities. Resources. Goals.
- Measures/Countermeasures: Courses of Action (COAs)

The system data can be modeled with widely-used architecture views that capture component and connector information, annotated with where data is stored, processed, and networked. User credentials can also be modeled here as attributes associated with components [30]. Attack scenarios can be modeled as exploits on vulnerable components, capturing data and credentials, with attackers moving laterally through a system via the connections. Functionality can also be modeled by leveraging industry-standard enterprise architecture views to represent functional hierarchies and dependencies that functions have on other aspects of the system architecture. The COAs are actions that change the system's technical architecture, including attributes of components or connectors in the architecture. An architectural level of abstraction works well for modeling information required by our approach.

3.7 Architectural Representations

We use a specific set of architectural views and their associated styles to capture the knowledge needed to model graceful degradation. Views are architectural representations of components of a system model and their relationships, while styles place constraints on those components and relationships. For example, in our running example, the Allocation View shows how the various networked components are connected in an instance of a transmission grid support network; the style constrains the network connections so that network services must connect to a switch or router (i.e., they cannot connect directly to each other).

First, we discuss the rationale for the inclusion of these specific views. The views included are explained in the following subsections in this order: 1. Functionality, a hierarchical understanding of how system functions contribute to higher-level functionality; 2. Component-and-Connector (Data Flow), including producers and consumers of data; and 3. Deployment (or Allocation), the runtime layout of components within the system.

3.7.1 View Selection Rationale

Happily, much of the data we need to satisfy our requirements described in Section 2.2 for automating evaluation of graceful degradation can be represented using three architectural views that are commonplace in either the software architecture or enterprise architecture fields. The table below shows how the key elements for our approach are modeled in the three architectural views we have selected.

Key Elements	Architectural View
System Data in Transit	C&C
System Data at Rest or in Transition	Deployment
Functionality	Functional
Credentials	Deployment
Attacker Capability	N/A (Parameter)
Attacker Points of Presence	Deployment
Defender Countermeasures	Deployment
Defender Budget	N/A (Parameter)

The Deployment View describes the layout of components and their connectors in a network, and it can be used to map out the possible attack traces. The attacker exploits vulnerabilities that exist on the components, and she then moves across the connectors to other components as she extends her attack trace. Her points of presence with the probability that she is there may be annotated on this view. However, her estimated capability for attack is not something represented in our selected architectural views; instead, it is represented as a parameter elsewhere in the evaluation.

The Component-and-Connector View is similar to a Deployment View, but it abstracts away some details so the focus can be on data producers, data consumers, and the data flows between the two. Data flows represent the transmission of data from a source to a destination.[2] A data flow does not imply an established "connection," as connectionless protocols like UDP transmissions or DNS queries can also be represented as data flows. In our approach, we usually refer to the source of the data as the producer and the destination of the data as the consumer. The data transmitted is the flow.

To concretely show the difference between a Deployment view and Component-and-Connector (C&C) View, in an enterprise network the Deployment View will show switches and routers, while the C&C View abstracts these details away simply showing producer components, consumer components, and the relationships between producers and consumers.

[2]In network monitoring, a data flow is usually defined by the transmission of data from a source IP address / port pair to a destination IP address / port pair, plus the actual data that is transmitted between the two.

The Functional View is a view showing the functions of a system and how they contribute to other functions in a hierarchy of nested functional requirements. That is some functions require other functions, which in turn require other functions. In this view, we integrate our components and data flows with their corresponding functions.

These views are interrelated. Functionality in a system requires data flows securely from producers to consumers. The producer-consumer relationships are represented in the C&C View. We determine if data flows securely by finding any overlaps of data flows (between producers and consumers) and attack scenarios. If an attacker compromises a component critical to a data flow, then the data flow is no longer secure. The attack scenarios and paths for data flows both exist in the Deployment View. Through evaluation across these three views, we are able to determine the impacts of attack scenarios on system functionality so we can evaluate an architecture for the purposes of graceful degradation.

For each type of system, architecture styles are also required. Recall that styles prescribe a vocabulary of system elements and relations between those elements together with constraints and rules about how those elements are composed. Using the Deployment View as an example, in a typical network, services connect to a router or switch; in a spacecraft, multiple components may directly connect to each other, or they may use a bus. Therefore, the constraints on how components connect – defined in the style – may vary depending on the type of system. These styles ensure that architecture alternatives conform with relevant architectural restrictions on those systems, and the styles can also constrain the number of valid alternatives.

As these are rules about the constraints in architectures, only a SME with deep architectural understanding can describe the style correctly. Styles are reusable between systems of similar types. Architecture styles are independent of system implementation and can be completed prior to design or drawn from a library of predefined styles. For example, a single style would likely be reusable between satellites, while another style would likely be reusable between electrical industrial control systems.

This selection of views is supported by both National Institute of Standards and Technology (NIST) and Consultative Committee for Space Data Systems (CCSDS). NIST states, "Three predominant viewpoints of system security include system function, security function, and life cycle assets. These viewpoints shape the considerations that are used as trustworthy secure design considerations for any system type, intended use, and consequence of system failure."

The NIST security function views include *passive* and *active* aspects. Passive aspects "include the system architecture and design elements." This is analogous to our use of the Deployment View. Active aspects are *behaviors* of the system and "include engineered features and devices, referred to as controls, countermeasures, features, inhibits, mechanisms, overrides, safeguards, or services." As they are behaviors, we represent them as tactics and COAs rather than in static views.

The NIST system function view is “the predominant viewpoint and establishes the context for the security function and the associated system life cycle assets.” It is “the system’s purpose or role as fulfilled by the totality of the capability it delivers combined with its intended use.” This corresponds with our Functional View. NIST states “the purpose of a system is to deliver a capability.” To this end, it proposes a view called a “Model for a System and Its Elements.” This hierarchical decomposition tracks closely with our approach to a Functional View.

The NIST life cycle assets include intellectually property to be protected, data flows within the system, and data dependencies between the system and the external environment. We capture these in our C&C View.

3.7.2 Functional View

The key to graceful degradation is the ability to evaluate trade-offs. In the case of graceful degradation for security, we evaluate the trade-offs between security and functionality. We are specifically concerned with the ability of each function to consume its associated data flows while meeting that function’s needs for confidentiality, integrity, and availability. We focus on data flows rather than components because, with rare exceptions, the primary target of hackers and the resource defenders protect is the data – not the hardware.

First, graceful degradation is about trade-offs At the highest level, a system performs functions. The functions are what give the system its utility, and – when under attack – some functions are sacrificed in a deliberate manner to preserve other functions. A hierarchy of functions allows us to evaluate trade-offs in a nuanced manner. We can also evaluate the cascading effects of various COAs on system functions. This is why the information contained in a functional view is necessary for evaluating graceful degradation.

Functional views are well-understood in the field of enterprise architecture [35], and we leverage this prior work and refine it for use in software architecture.

At the highest level, our functional view is a logic diagram of AND and OR operators to define the relationships between functions and sets of sub-functions. Some functions have utilties associated with their completion, while others do not. Only functions that provide value on their own have an associated utility. Just like an avionics system provides no value when not installed in an aircraft, an HMI provides no value when not part of a functioning electrical transmission system. Functionality provides utility when the functionality’s required components are not compromised and the functionality’s data flow consumers can retrieve necessary data from producers with pre-specified security attributes (i.e., confidentiality, integrity, availability).

At the leaves of the upside-down tree are associations between functions and dataflows. The leaves connect a function or sub-function to a consuming component’s ability to consume a dataflow securely.

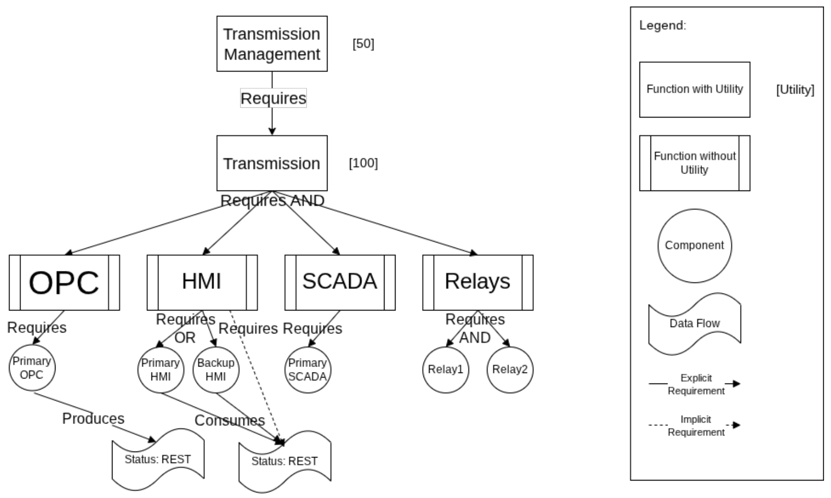

Figure 3.2: Functional View example.

A function that is a *consumer* of a data flow determines the confidentiality, integrity, and availability requirements for its associated data. A component that is a *producer* of data may offer some guarantees (e.g., the data is cryptographically signed, so the integrity is assured), but it the consumer (and its associated function) that establishes the consumer's requirements.

Consider the metaphor of weather forecasts: The weather service may provide the local forecast without knowing all of the possible consumers of the forecast or all of the possible ways in which the forecast is used. A civilian consumer of weather data may not be concerned with confidentiality, but a military consumer can be if the weather data provides a combat advantage. Both consumers will be concerned about the integrity and availability of the data. Thus, the onus for requirements on data flows is placed on the consumer of the data.

An example of part of a Functional View for the running example (introducted in Section 3.5) is shown in Figure 3.2. In this example, the HMI function serves to represent the purpose fulfilled by either a Primary or Backup HMI component. These two components consume system status data (in the REST format) on behalf of the HMI function. In our example, the transmission management functionality requires the transmission functionality (or there would be nothing to manage), and the transmission functionality requires the human machine interface service.

3.7.3 Component-and-Connector (Data Flow) View

A data-centric approach is advantageous over approaches that consider only the protection of components. Functions require secure storage, processing, and transmission of data. Data is transmitted from a producer to a consumer. A Component-and-Connector View is an architectural model that highlights the relationships between data producers and consumers. The C&C

View abstracts away details such as the specific paths the data takes through the Deployment View. It represents only data flows, data producers (and associated security requirements), data consumers (and associated security guarantees), and their relationships.

> A C&C view shows elements that have some runtime presence, such as processes, objects, clients, servers, and data stores. The elements are called *components*. Additionally, component-and-connector views include as elements the pathways of interaction, such as communication links and protocols, information flows, and access to shared storage. Such interactions are represented as *connectors* in C&C views. [30]

In our implementation of C&C views, we focus on the consumers and producers of information, and we hide (abstract away) elements that do not produce or consume the data flows relevant to our models, such as routers, switches, and firewalls.

Figure 3.3 depicts a component-connector view of the running example system. While the C&C View does not represent some details of the Deployment View like switches and routers, it does represent details like which component is producing a data flow and which one is consuming it. This provides insight into the flow of data at a more logical, as opposed to physical, level. Data flows are the glue that binds together the architectural views for our evaluations of secure graceful degradation.

3.7.4 Deployment View

With our approach, we are able to dynamically consider how alternate paths for data flows can optimize functionality in the face of attacks and graceful degradations. To do so, we must integrate the prior views with one that shows exactly how data flows through the system and also how the attacker moves through the system. Because the C&C View is focused on the producers and consumers of the data flows, it abstracts away the intermediate components through which data may flow and across which attackers move.

Attacks can originate from multiple points of presence [36], and we represent these in the Deployment View. Each system component has an associated probability of being compromised and becoming an attacker point of presence. By representing attacks from multiple points in this way, we can represent Black Swan (low probability, high impact) events [37]. Not only can we represent probable attacks (e.g., originating from the internet), we can simultaneously represent improbable high impact events like insider threats. Without this simultaneous representation, our approach would need to choose between protecting from one threat or the other; with a simultaneous representation, we do not need to ignore the low possibility of a Black Swan event. This also allows us to explain the usefulness of tiered architectures. Rather than a monolithic perimeter with all the security expenditure in one place, a tiered architecture distributes security

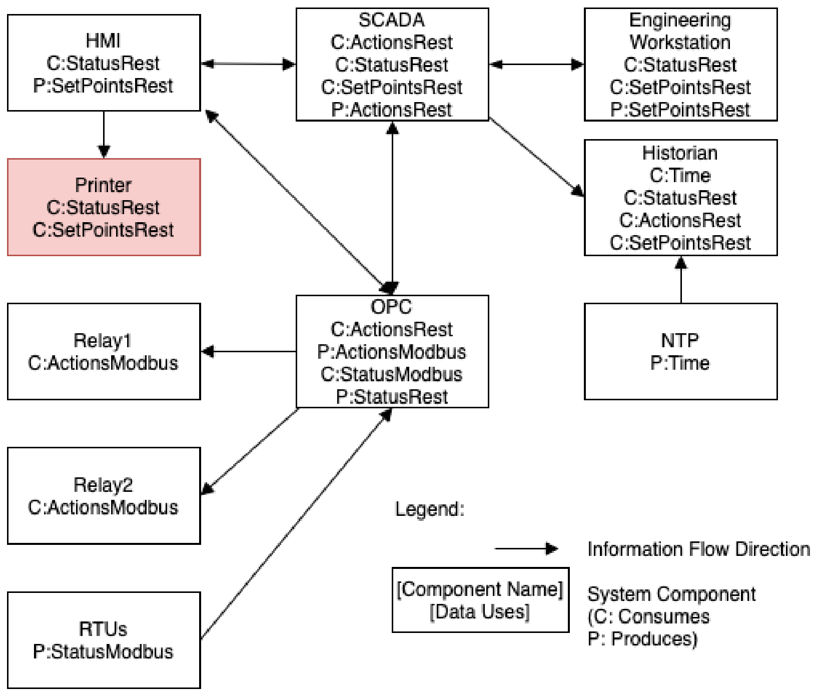

Figure 3.3: Component-and-connector view with data flows of the running example ICS system.

through multiple perimeters separating tiers. An attack attempt originating from the Internet is probably much more likely than an insider attack; however, an insider attack could be devastating if there are no precautions in place to protect against this scenario.

Elements of a Deployment View can include software and hardware. Software is allocated to specific hardware components, and network connections provide for data transmission between components [30]. The deployment view does not represent information flows, but it does include components through which data flows like network routers and switches that are abstracted away (hidden) in a C&C View.

Figure 3.1 depicts a deployment view, showing the physical (OSI Layer 1 and 2) connectivity of the system components. This view is useful in determining how an attacker can move laterally.

In our model, we also have a notion of credentials. These can be passwords, session tokens, or anything similar that can be stolen and reused. Components may have or use credentials. If an attacker exploits a component that has a store of credentials, she gains access to those credentials, which she may use elsewhere just like an exploit to gain access to other components that use those same credentials for access. These have been vital to the propagation of attacks in the real world, and our model includes them. A component can be "exploited" by the unauthorized use of a credential.

3.8 Attacker Representation

As specified in the requirements in Section 2.2, we choose to represent the attacker as omniscient about the defender's system. In other words, we model attackers' capabilities and points of presence in a way that represents uncertainty about the possible attack scenarios. To address uncertainty about attacker goals, we do not assume a particular goal for the attacker; instead, we use the worst case attack scenarios in our evaluations. Just as there may be uncertainty about the location of the attacker as discussed in Section 3.7.4, there may be uncertainty about the capability of an attacker. Here, too, we must consider Black Swan events if we wish to avoid being the victim of them – in this case, we can represent the unlikely possibility an attacker has a high level of capability. This ensures that the defender does not simply take the expected capability of the attack and rest comfortably, with a 50% probability of being caught off-guard. And by allowing for multiple points of presence, we ensure we capture the possibilities of multiple attackers or a single attacker leveraging multiple simultaneous attacks.

The attacker moves through a system by expending capability to develop exploits against components represented in the Deployment View. Exploits act on specific component types (and the subtypes of that type). They have user-definable impacts on confidentiality, integrity, and availability. The first use of an exploit comes with a cost to the attacker's capability. This represents the initial cost to develop an exploit in the real world. The connectivity between components in the Deployment View constrains how an attacker moves through a system.

As an attack progresses, the attacker can reuse exploits on components of the same type without an additional marginal cost of exploit development. In reality, there may be a small additional cost to reuse an exploit against a new target (e.g., customizing scripts to point to the new target IP address), but this is far smaller than the cost to develop an exploit for a given vulnerability, so we simplify by treating the cost of reusing an exploit as zero. Additionally, the attack scenario may look more like a tree than a single line from Point A to Point B. This is important when evaluating how a backup system can contribute to graceful degradation. If a backup subsystem is a clone of the primary subsystem, the attacker may be able to branch the attack scenario and compromise both the primary and backup systems with little or no additional cost beyond compromising just the primary system. Therefore, we need to represent types of components (e.g., Windows 11), reuse of exploits, and branching attack scenarios.

Beyond exploits of vulnerabilities, we must also represent credential theft and reuse. This allows us to evaluate attack TTPs like those found in the Solar Winds attack. If a component is compromised, some credentials stored on the component may be compromised, too, and these can be used for lateral movement or privilege escalation elsewhere in the attack scenario.

3.9 Defender Representation

We must also represent how our active defender may react to an attack scenario to gracefully degrade the system. First, we introduce some basic assumptions about the defender. Then we discuss the types of tactics that could be available to the defender and evaluable with an approach such as ours.

We assume our defender has a budget that he can spend on defense. This limits his ability to make defensive changes to his system. However, the cost of changes will be different at architecture and design time versus run time. The graph edit distance approach also limits the search space for alternatives. For example, the "cost" to add a connection or remove a connection is incurred by the defender each time one of these tactics is applied.

The types of defensive tactics we can represent correspond closely with what we represent in our model. These tactics include:

Adding a Connection Inserts a direct network connection between two components.

Removing a Connection Deletes a direct network connection between two components.

Adding a Component Using a component that already exists in the model but is isolated from the rest of the network, add a connection to connect this component to the larger network.

Removing a Component Remove all connections to a component.

Adding a Credential Add a new credential to a component for storage on that component or use by it.

Removing a Credential Remove an existing credential that is stored on or used by a component.

Changing a Credential Change the credential that a component uses (by removing one and adding another in its place).

These defensive tactics can emulate key tactics used as defense in real attacks.

3.10 Uncertainty Representation

We address uncertainty throughout our approach. For the system, we represent hypothetical vulnerabilities with estimated costs to exploit. For attackers, we use estimated capabilities and points of presence with associated probabilities of compromise. There are some slight differences in how uncertainty is represented in architecture and design time versus run time. In this section, we begin with a discussion of architecture and design time considerations for uncertainty. Next, we describe how we address run time uncertainty. Finally, we discuss how the uncertain values of parameters can be estimated.

At architecture and design time, many of the implementation decisions have not yet been made, so we cannot assume, for example, that a system will have a particular operating system version with a database from a specific vendor and with a specific version. However, we will know that there will be an operating system and a database, and any associated values (e.g., cost to exploit) are estimated. For subsystems that have not yet been designed and decomposed into constituent components, we use architectural abstraction to hide the undecided details of the defended system early on. For example, a subsystem can be represented as a single component rather than a set of components.

Also at architecture and design time, we assume the defender does not know the exact identity, specific capabilities (i.e., exploits, budget), or goal of our attacker. However, the defender might partially understand and make assumptions about the threats against the system. For example, a bank or gambling site might be subject to attacks by organized crime seeking to steal money. An aerospace company might be subject to attacks by a nation state seeking to understand the manufacturer's state of the art and steal its intellectual property, though the specifics might not be understood. We can still bound the attacker based on relative levels of capabilities. Additionally, we might know at design time where the attacker is likely to gain presence on a system.

At run time, we assume that system administrators are unlikely to leave known vulnerabilities unpatched for long periods of time. As responsible administrators, they will patch systems as soon as possible. If a patch is unavailable, other mitigations might be an effective – if temporary – solution. Therefore, our attacker model still relies on the use of unknown vulnerabilities.

At run time all estimated values – cost of exploit, attacker capability, and points of presence – can be updated. For example, when there is clear evidence of an attack, the probability of attacker presence on a particular component can adjusted to be at or near 1.0 (100%). These updates trigger a re-analysis of the architecture to identify options for graceful degradation.

It is possible to estimate these values at both architecture and design time and at run time. Attacker capability is determined by SMEs as part of a threat assessment. In our research to date, the capability is represented by the number of zero-day attacks[3] an attacker has. However, this capability could also represent a monetary equivalent that an attacker has access to, with each zero-day vulnerability reducing the remaining attacker capability.

SMEs for threat intelligence can leverage historical data to inform attacker capability estimates. For example, the sophisticated Stuxnet attack on Iranian centrifuges used four zero day vulnerabilities [38]. Though the cost to create these zero days is not publicly known, other

[3]A zero-day attack is one in which an exploit is used against a vulnerability that is unknown to the defender.

sources have provided black market sales prices for zero days [29]. This sale information can be used by SMEs who must estimate the vulnerability (difficulty of exploiting) component types. Other data sources include attack surface estimates [39], penetration testing data, and historical vulnerability data [40]. Some zero day exploits are sold to only one customer, while others are sold to multiple customers, amortizing the cost of exploit production. This information can be used to generate rough order of magnitude estimates with which to assign a cost to exploit a particular component type.

3.11 Process Diagram, Information Requirements, and Burden

The information that must be gathered is summarized in the table below. The most costly information is that which is gathered by a SME, not automatable, and mutable (i.e., it can change through that part of the system life cycle). Data requirements are described for both architecture and design time and run time.

Data	Source	SME	Automatable	Mutable
Deployment View (D*)	Architecture	Y	N	Y
Deployment View (R**)	Scanning	N	Y	Y
Deployment Style (D/R)	Architecture	Y	N	N
Component-and-Connector View (D)	Architecture	Y	N	Y
Component-and-Connector View (R)	Scanning	N	Y	Y
Component-and-Connector Style (D/R)	Architecture	Y	N	N
Functional View (D/R)	Architecture	Y	N	N
Functional Style (D/R)	Architecture	Y	N	N
Attacker Representation (D/R)	Threat Sources	Y	Partial	Y
Defender Representation (D/R)	Internal Sources	Y	N	Y

* (D) Architecture and Design Time ** (R) Run Time

Our approach integrates the architectural views with the attacker and defender profiles, evaluating the information contained within them. The output of the approach is an optimal defensive COA for this model of the system and environment. The entire process – with inputs and outputs – is captured in Figure 3.4.

The process begins with an initial architecture. The architecture is provided by the architect (at design time) or system administrator (run time) in the form of an Allocation View and architecture styles that constrain the architecture. The implementation of our approach produces a number of architecture alternatives for consideration; these alternatives are constrained by the defender profile.

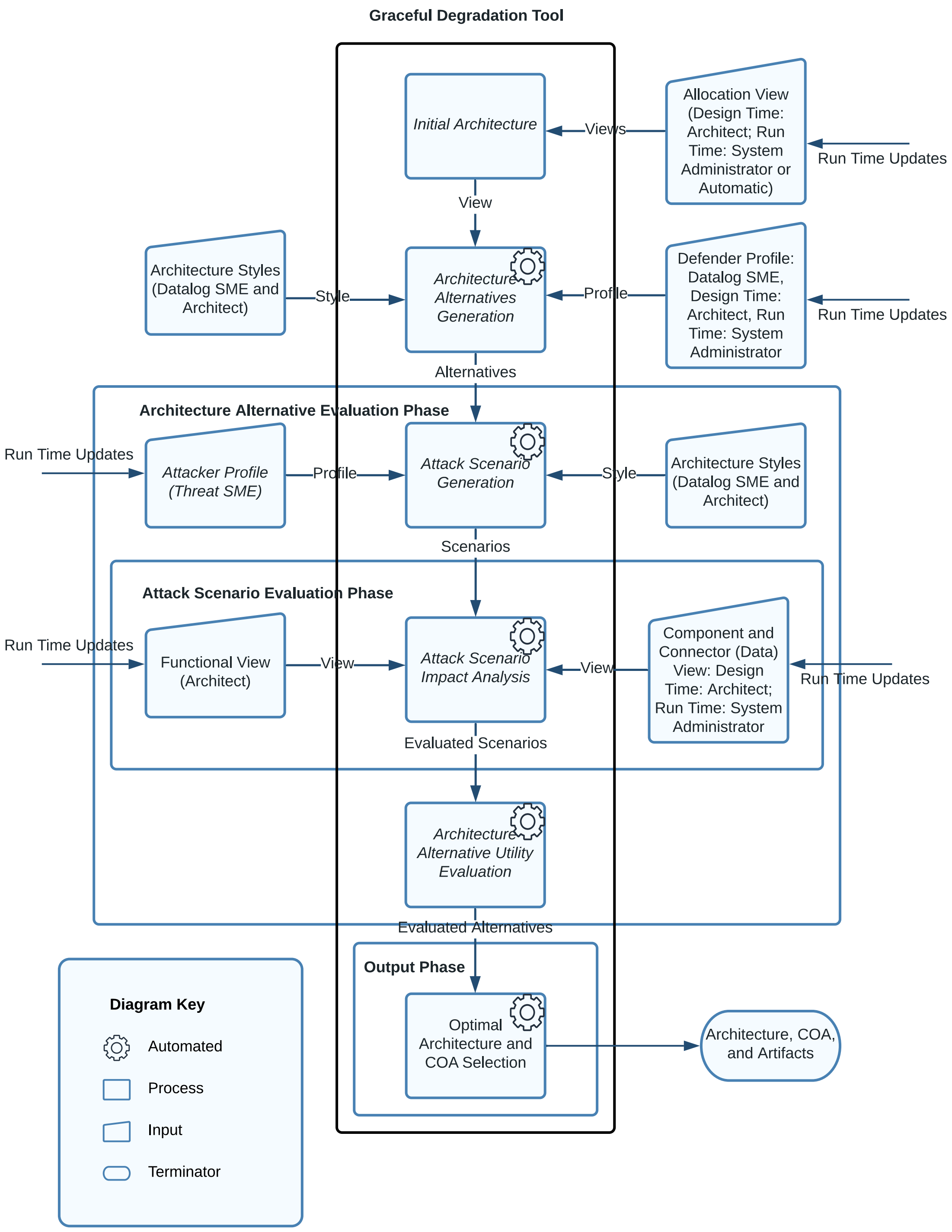
Graceful Degradation Tool
Initial Architecture
Views
Allocation View (Design Time: Architect; Run Time: System Administrator or Automatic)
Run Time Updates
View
Architecture Styles (Datalog SME and Architect)
Style
Architecture Alternatives Generation
Profile
Defender Profile: Datalog SME, Design Time: Architect, Run Time: System Administrator
Run Time Updates
Alternatives
Architecture Alternative Evaluation Phase
Run Time Updates
Attacker Profile (Threat SME)
Profile
Attack Scenario Generation
Style
Architecture Styles (Datalog SME and Architect)
Scenarios
Attack Scenario Evaluation Phase
Run Time Updates
Functional View (Architect)
View
Attack Scenario Impact Analysis
View
Component and Connector (Data) View: Design Time: Architect; Run Time: System Administrator
Run Time Updates
Evaluated Scenarios
Architecture Alternative Utility Evaluation
Evaluated Alternatives
Output Phase
Optimal Architecture and COA Selection
Architecture, COA, and Artifacts
Diagram Key
Automated
Process
Input
Terminator

Figure 3.4: Concept of Operations process diagram with stakeholders, data inputs, and data outputs.

Each of the alternatives is then evaluated for its resilience to attacks. The attacker profile (provided by a threat analyst SME) and architecture styles constrain the attack scenarios that are produced. The set of attack scenarios for the architecture is overlaid on the system's data flows to identify where the attack scenarios may compromise data flows; the data flows are provided in the C&C View. The Functional View contains the information necessary to determine how the compromise of data flows impacts functionality and thus utility. A utility is evaluated for each possible attack scenario of each architecture alternative.

Our approach evaluates an estimated utility for each alternative based on the probability of each combination of attacker initial conditions (i.e., presence and capability). The worst case attack scenario for a combination of attacker initial conditions is considered the representative attack scenario for those initial conditions. We compare the alternatives based on their estimated utilities.

The alternative with the highest estimated utility is the recommended architecture of our approach. This architecture, the COA to transition from the initial architecture to the recommended architecture, and evidentiary artifacts (i.e., attack scenarios and evaluations) are produced as outputs in the process.

3.12 Tooling

To prove the concept of our approach, we implemented a tool called Defensive degradation of Resilient Architectures (DORA). This tool demonstrates how careful modeling of systems (via architecture views and styles), attackers, and defenders – combined with a formal approach to evaluation – can be used to understand systems' levels of security and how they can gracefully degrade in response to attacks. In the following chapters, we describe the implementation of DORA and our evaluation of it.

Chapter 4

Implementation

In Chapter 3, we introduced our approach and described high level design decisions. To demonstrate the concepts of our approach, we implemented a tool we call Defensive degradation of Resilient Architectures (DORA). The implementation details of DORA are described in this chapter. First, we explain our reasoning for the specific programming languages that DORA is implemented in. Next, we explain how we represent architecture views and styles. After that, we show how the attacker and defender are represented with respect to the architecture views. Finally, we explain how we implement the generation of architecture alternatives, create the attack scenarios within those architectures, and evaluate the graceful degradation security properties of each architecture.

4.1 Programming Languages

We chose Python and PyDatalog as the primary tools to implement our approach. Python is an interpreted language that is widely used, well-supported, easy to program, and has integrations with many third-party tools, resulting in great extensibility. PyDatalog is a Python-based tool that evaluates Datalog, the language we chose for representing and evaluating architecture alternatives in various attack scenarios [41].[1]

Attack scenarios occur step by step. Attackers do not teleport around systems – they must move to adjacent components or networks by exploiting components one after another. Similarly, the systemic implications of a loss of particular functions can be determined through inductive reasoning. Declarative logic programming languages are ideal for these kinds of analyses.

Datalog is a declarative logic programming language related to Prolog. It has many useful attributes: Simple *facts* and *rules* (logical implications) are easy to read and write. We induc-

[1]PyDatalog is no longer actively supported. However, the concepts in this paper are generalizable to other implementations of Datalog and even other declarative logic programming languages.

tively evaluate the rules to build attack scenarios, which are at the core of how our analysis works. We also use Datalog rules to reason about how data flows through the system and the implications of various attack scenarios on system functionality. Datalog has a successful track record, having already been used with MulVal [11], which we discuss in Section 2.3.3.

We considered alternatives to pyDatalog. Tools such as PRISM and Alloy solve problems differently. For example, PRISM is a probabilistic model checking tool that answers questions like "'what is the probability of a failure causing the system to shut down within 4 hours?', 'what is the worst-case probability of the protocol terminating in error, over all possible initial configurations?', 'what is the expected size of the message queue after 30 minutes?', or 'what is the worst-case expected time taken for the algorithm to terminate?"' [42][43] It addresses uncertainty well but does not have a straightforward means to generating attack scenarios or data flow paths. Alloy leverages satisfiability solvers to find solutions to problems with constraints [44]. While we are trying to find solutions within constraints, graceful degradation requires finding an optimal (or near optimal) solution given uncertainties. Secure graceful degradation requires us to be able to sequentially build out attack scenarios. Alloy is also not well-suited to build out attack scenarios and calculate utility. For these reasons, we found that Datalog (implemented in pyDatalog) was likely to be the most straightforward tool to implement our approach.

4.2 Views and Styles

In our tool, we separate architecture views and styles into separate files. Styles are constructed through a set of Datalog rules. Styles are reusable for architectures of similar types of systems (e.g., enterprise networks, satellites, etc.). They are not likely to change during the course of a system life cycle. A Subject Matter Expert (SME) with Datalog and architecture expertise can define the style.

Views can be constructed through a set of Datalog facts. They are system-specific and change to reflect modifications to the system architecture over the system's life cycle. Views are more straightforward to define than styles, so they can be defined by a system architect (at architecture and design time) or system administrator (at run time), and they are amenable to integration with automated tooling.

4.2.1 Deployment View in Datalog

To implement our running example in Datalog, we begin with our Deployment View, depicted in Figure 3.1. First, we define the components that comprise the system. For example, we define that the email server is of type "emailServerT," and SwitchA and SwitchB are network switches:

```
+ isType('emailServer','emailServerT')
+ isType('switchA','switch')
+ isType('switchB','switch')
```

We also need to define the vulnerability of the components. The following line of code defines that the emailServerT type (email server type) component takes one unit of attacker capability to exploit, and it zeroes out any confidentiality, integrity, and availability guarantees of the component:

```
+ isVulnerable('emailServerT','emailServerExploit',1.0,0.0,0.0,0.0)
```

We can create type hierarchies, which are useful for exploits that might be shared across various subtypes of components. This is useful for components made by a common vendor that run slightly different versions of software. In these cases, the components may have common vulnerabilities. For example, we can specify that a corporateFW component is an enterpriseFirewall1 type, and an enterpriseFirewall1 type is also a firewall type.

```
+ isType('corporateFW','enterpriseFirewall1')
+ isSubType('enterpriseFirewall1','firewall')
```

In this example, any exploit that is against the firewall type will apply to the enterpriseFirewall1 type, which includes the corporateFW component.

In addition to creating exploits, attackers can guess or steal credentials such as passwords and session tokens. An attacker accumulates credentials from a component when that component is compromised. Components can have more than one credential that the attacker can steal. Just like an exploit, a credential – once available to the attacker – can be reused with no additional "cost" to the attacker (i.e., to the attacker's capability). We represent a credential on a component with a Datalog fact. In the following example, the corporate firewall component has the firewall password and a network management session token on it. If the attacker compromises this component, the attacker gains access to both credentials. The firewall password can be used to access the corporate firewall, and the management session token can be used to access both the corporate firewall and the email DMZ component.

```
+ hasCredentials('corporateFW',['fwPassword','mgmtToken'])
+ usesCredential('corporateFW','fwPassword')
+ usesCredential('corporateFW','mgmtToken')
+ usesCredential('emailDMZ','mgmtToken')
```

In addition to defining the components and their attributes, we define the connectors across which data and attackers move. Connectors indicate a relationship between two components. In this case, the connector indicates that the two components can communicate directly with each other. Each connector also provides confidentiality, integrity, and availability guarantees as specified in the Datalog fact that defines it.

The Datalog fact below defines a (bi-directional for this type of system) connection between the corporate firewall and the network device that establishes an email DMZ ("de-militarized zone," a subnetwork with a trust level between other subnetworks in a system). The three boolean values that follow the component names define the extent to which this connection guarantees confidentiality, integrity, and availability. In this case, all three are fully guaranteed.

```
+ connectsTo('corporateFW','emailDMZ',1.0,1.0,1.0)
```

The deployment view also has associated architectural style rules. One example is a rule that a service (e.g., web service, mail service, file service) on the network must connect to a network device (e.g., router, switch) rather than directly to another network. The rule states that, if a service type connects to a switch type, that is a valid network connection.

```
validNewConnectsTo(SourceService,TargetService) <= isType(SourceService,'
    switch') & isType(TargetService,'service') & ~(SourceService ==
    TargetService) & ~connectsTo(SourceService,TargetService)
```

Listing 4.1: Networking Style

4.2.2 Component-and-Connector View in Datalog

In the simplified version of the Component-and-Connector View for our electrical system in Figure 3.3, we see how the OPC component produces status data in the REST format. For the purpose of supporting the electrical transmission function (functions are described in Section 4.2.3), the SCADA server and HMI consume this status data flow; they do not require confidentiality, but integrity is twice as important as availability. The associated facts represented as Datalog predicates are shown below:

```
+ producesData('opc','statusRest')
+ consumesData('transF',['scada'],'statusRest',0,0.67,0.33)
+ consumesData('transF',['hmi'],'statusRest',0,0.67,0.33)
```

Listing 4.2: Data Flows

Note how the Component-and-Connector View differs from the Deployment View. It shows data producing and consuming components with their associated data flows while abstracting away intermediate nodes like firewalls and switches.

4.2.3 Functional View in Datalog

The Functional View depicts the various functions that hierarchically contribute to each other and the system's overall utility; evaluation across these functions is how we ensure that changes to a system provide graceful degradation rather than indiscriminate effects. The Functional view is a logical hierarchy that expresses composition/decomposition of functional requirements:

Leaf nodes. These connect data flow consumer nodes to functions based on the consumes statements we describe earlier in this chapter.

Intermediate nodes without utility. These nodes are either logical AND or logical OR nodes. while these nodes are part of the functional hierarchy, they do not, by themselves, represent a subset of functionality that has a utility. Consider the case of a web server front end intermediate node. The servers cannot perform their roles without the back end intermediate node functionality working in tandem. So, a web server front end node would have no utility associated with it.

Root and intermediate nodes with utility. These nodes are also logical AND or logical OR nodes. However, these nodes represent a subsystem that by itself can provide some functionality. Continuing on the example above, a web service intermediate node might require the front end intermediate node AND the back end intermediate node to both indicate secure functionality so the web service is securely functional.

We implement elements of the Functional View shown in Figure 3.2 in Datalog.

First, some functions directly contribute to overall system utility. For example, the Transmission function has a utility of 50, as defined below:

```
+ utility('transmissionF',50.0)
```

Listing 4.3: Utility Definition

In our example, the Transmission Management functionality requires the Transmission functionality (or there would be nothing to manage), and the Transmission functionality requires the Human Machine Interface (HMI) service. This is defined in Datalog as:

```
+ fNodeAnd('transmissionMgmtF',['transmissionF',['opcF','hmiF','scadaF','
    relaysF','rtusF'])
```

Listing 4.4: Functional Relationships

We define that the HMI data production can be fulfilled by either a Primary or Backup HMI component.

```
+ producesData('hmiPrimaryServer','setPointsRestData')
+ producesData('hmiBackupServer','setPointsRestData')
```

Listing 4.5: Backup Data Producer Representation

Functions may require data flows with specific security attributes. Note that the above two components consume system status data (in the REST format) on behalf of the HMI function. The Supervisory Control And Data Acquisition (SCADA) server consumes the set points data (in REST format) as shown in the code below; integrity and availability are equally important, while confidentiality is not important. Either component that produces the set points data in REST format can be a source of data for the SCADA server. The consumption of data flows is common to both the Component-and-Connector (C&C) and Functional Views, and it binds those two views together.

```
+ consumesData('scadaF',['scada'],'setPointsRestData',0.0,0.5,0.5)
```

Listing 4.6: Data Consumption Representation

4.3 Attacker and Defender Profiles

4.3.1 Attacker Profile

The attacker is represented by points of compromise and probability of compromise. In the following example code, the attacker has a 90% probability of having a presence on the “internet” component, which is an abstraction representing the internet. If the attacker has presence there, she zeros out (since False is equal to 0.0) the confidentiality, integrity, and availability guarantees respectively of that component. She also has a 10% probability of appearing as an insider on a corporate owned workstation, with similar effects to that component. In our assessments, the possibility of an insider threat is a crucial assumption that informs the rationale for tiered architectures.

We define our points of possible compromise like so:

```
+ compromised('internet',0.9,False,False,False)
+ compromised('businessWorkstations',0.1,False,False,False)
```

4.3.2 Defender Profile and COAs

A defender can modify the system through application of pre-defined tactics. Each sets of tactics is referred to as a Course of Action (COA). Each tactic has a corresponding cost associated with it. The tactics that can be applied are bounded by the defender’s budget, implemented as a numerical parameter in Python. Tactics are applied to the Datalog model via Python code that adds or removes facts from the system.

Tactics can address the architecture directly. For example, adding and removing connections can change the topology of a network. These tactics can also result in the addition or removal of components from a network. In our implementation, all components exist at the outset of the evaluation, though they might not all be connected into the system being defended.

4.4 Model Evaluations

With a method to model the defended system, attacker, and defender we have the ability to create related architecture alternatives, simulate attack scenarios, and calculate and estimated residual utility (i.e., utility after accounting for attack) for each alternative. In the following subsections, we describe each of these steps.

4.4.1 Generation of Architecture Alternatives

Architecture alternatives are generated in Python. These begin with an initial architecture that is pre-specified. Sets of tactics, or COAs can be applied by the defender to change the architecture from the initial architecture to an architecture alternative. The cost of a COA acts as a graph edit distance from the initial architecture. A pre-defined constraint of the defender's budget acts as a constraint on the number of possible architecture alternatives at run time. At architecture and design time, this constraint limits the search space to architectures similar to the initial architecture. We do not create and evaluate all the alternatives in PyDatalog at once; rather, we evaluate them one at a time.

4.4.2 Attack Scenario Generation

Our tool inductively applies Datalog rules to construct attack scenarios. A scenario originates at a compromised component. It extends one move at a time through the exploitation of vulnerabilities, use of credentials, or movement where no exploit is required (e.g., moving through a network switch). The scenario terminates when the attacker has expended her budget, as defined by her capability, or has no other possible moves. The attack scenario generation code snippet is also found in Appendix C.

In the code below, we see a Datalog rule for an inductive case to add an edge to an existing scenario. The scenario is defined by where it begins (SourceService), where it ends (TargetService), a path, a list of exploits used so far, a list of attacker moves (i.e., use of a particular exploit against a particular component), and the cost of the trace to the attacker. If a current scenario exists, and that scenario termination can be extended via the use of a new exploit (shown as VulnType._not_in(E2)), then a new scenario is created and added to the Datalog logic database. The prior scenario remains in the database, also. The new scenario reflects the new termination point, new exploit, and additional cost of the exploit.

```
#Inductive case to continue connectivity, new exploit
attackPaths(SourceService,TargetService,P,E,AttackerMoves,TotalC) <=
    attackPaths(SourceService,IntermediateService1,P2,E2,AttackerMoves2,
    TotalC2) & cToWithPrivileges(IntermediateService1,TargetService,VulnType
    ,C) &
(P==P2+[IntermediateService1]) & (VulnType._not_in(E2)) & (E==E2+[VulnType
    ]) & (TotalC==TotalC2+C) & (TotalC2+C <= MaxR) & (AttackerMove==[
    IntermediateService1,TargetService,VulnType]) & (AttackerMove._not_in(
    AttackerMoves2)) & (AttackerMoves==AttackerMoves2+[AttackerMove]) & (
    SourceService!=TargetService) & (SourceService._not_in(P2)) & (
    TargetService._not_in(P2))
```

In the case an attacker reuses an existing exploit (reflected in VulnType._in(E2)), the cost of the scenario does not increase (shown as TotalC==TotalC2). This is shown in the code below.

```
#Inductive case to continue connectivity, previously-used exploit
attackPaths(SourceService,TargetService,P,E,AttackerMoves,TotalC) <=
    attackPaths(SourceService,IntermediateService1,P2,E2,AttackerMoves2,
    TotalC2) & cToWithPrivileges(IntermediateService1,TargetService,VulnType
    ,C) & (P==P2+[IntermediateService1]) & (VulnType._in(E2)) & (E==E2+[
    VulnType]) & (TotalC==TotalC2) & (TotalC2+C <= MaxR) & (AttackerMove==[
    IntermediateService1,TargetService,VulnType]) & (AttackerMove._not_in(
    AttackerMoves2)) & (AttackerMoves==AttackerMoves2+[AttackerMove]) & (
    SourceService!=TargetService) & (SourceService._not_in(P2)) & (
    TargetService._not_in(P2))
```

4.4.3 Calculation of Estimated Residual Utility

For each architecture alternative, we generate a residual estimated utility. The residual estimated utility represents the estimated value of the utility of the architecture alternative after accounting for our attacker. This is all evaluated with PyDatalog code.

To estimate this value, for each possible attacker capability, we determine the residual utility of the architecture alternative for that attacker capability. We multiply this value by the probability the attacker has this capability; the sum of these multiples is the estimated residual utility.

The residual utility of a specific architecture alternative with a given attacker capability is determined by inductively producing all possible attack scenarios for this attacker capability. The worst case scenario attack scenario (i.e., the one the results in the lowest utility) is used to represent that combination of architecture alternative and attacker capability.

Datalog can capture the worst case scenarios for a range of attacker capabilities using this code:

```
(worstCasePath[TotalC] == max_(UtilPathPair, order_by=U)) <=
    pathCompromisesWithCost(X,C) & (pathCompromisesUtilities[X] == U) & (
    pathCompromisesFunctions[X] == FList) & (UtilPathPair==[U,FList,X]) & (
    TotalC >= C)
```

The max_ function takes the worst case path, ordered by utility, of the UtilPathPairs, which capture the traces, affected functions, and resulting utilities.

This process of calculating residual utility does not assume any particular type of system (e.g., enterprise network, control system network, etc.). The rules are reusable for different types of systems.

Chapter 5

Validation

5.1 Definition of Validation Criteria

In this chapter, we describe the properties that we are evaluating and how they can be traced back to the claims made in the thesis statement in Section 1.3:

We can architect and operate systems that are better able to weather attacks by automating the evaluation of systems' security properties to enable effective automated graceful degradation of systems in the presence of uncertainty through an approach of formally modeling systems and system behavior at an architectural level of abstraction to explore hypothetical attacks and the systems' abilities to respond.

Important properties of this approach include scalability and performance, realism, usability, and effectiveness.

Some of these claims are evaluated through experimentation using case studies, whereas others are evaluated through qualitative arguments. The approach to evaluation depends on the type of property being tested. Where we can evaluate quantitatively, we do so. In cases in which we cannot (e.g., due to resource constraints or the qualitative nature of the evaluation), we evaluate qualitatively. In one case, we use an ensemble analysis,[1] making minor changes in the inputs and measuring changes in the outputs. The correspondence between the claim and the validation type is shown in Table 5.1.

Our validation applies to both the architecture and design time and run time use cases. The inputs are either estimated (architecture and design time) or mostly known (run time). The amount of time to evaluate architecture alternatives is most relevant at run time, when time is of the essence; however, Courses of Action (COAs) can be generated and stored in advance to anticipate attacks and ensure rapid responses.

[1]We borrow the term "ensemble" from the field of meteorology, in which an ensemble forecast is an average of multiple forecast runs using small variations to the initial conditions.

To evaluate the effectiveness of our approach, we use a combination of argument and case studies. This is because there is no agreed-upon ground-truth quantification of the security of a system. Similarly, Subject Matter Experts (SMEs) could bring their own biases toward modern conventions on secure architecture patterns, and we wish to avoid any preconceived notions in our evaluations. Our smaller case studies demonstrate the key properties of our approach in an uncomplicated manner. They demonstrate how they correspond with best practices for secure system architectures. Through larger examples, we demonstrate that these key properties continue to hold. We argue that effectiveness will continue to hold for larger, more complex examples.

Because of the amount of time and computation for each full run (i.e., including the consideration of multiple architecture alternatives) of our approach, we cannot evaluate large numbers of examples for a fine-grained quantitative assessment of how our approach scales. Instead, we use case studies of varying sizes (i.e., number of components, attacker points of presence, etc.) to estimate the scalability of our approach. By stepping through our approach and tool algorithmically, we argue how the computational and time requirements scale with respect to the size of the case study. Although there will be some variation, the scalability of our approach should be roughly comparable to an interpolation of the case studies we present.

Uncertainty is, by its nature, uncertain. We argue that our approach models relevant uncertainties that defenders face in real-world environments. We use a sensitivity analysis to show that our approach to modeling uncertainty provides value by anticipating a range of possible attack scenarios while not introducing so much uncertainty as to make the analysis generic (i.e., COAs rarely change) or unstable (i.e., COAs change dramatically with each small change to input).

Usability can be difficult to quantify, since it varies from one user to another based on their knowledge and experience. A study with human subjects could provide insight into how SMEs approach determining factors that are not easily quantifiable (e.g., probability of compromise of a component, required capability to compromise a component, attacker capability). We argue that other research shows that these numbers can be estimated by SMEs. We argue that our knowledge requirements are the minimum needed for an evaluation of secure graceful degradation and we make data collection and reuse as simple as possible and minimize the use of SMEs. We show through a sensitivity analysis that the numbers can vary some without large changes to evaluation output. Our effectiveness analysis shows that these rough estimates are sufficient for our approach to produce useful COA outputs.

Realism is a measure of our approach's ability to model relevant aspects of a real world system with its environment, attack scenarios, and defensive COAs. Realism does not require modeling all the details of the real world – just those that are most relevant to secure graceful degradation. In our validation, we evaluate three different types of systems of three different scales. These case studies demonstrate our ability to model relevant details in different contexts. We argue that this can be extrapolated to systems of various types and sizes.

In the following sections, we describe our evaluation approach. We define each claim, provide a measure or metric, describe our approach to determining the measurement, and explain our success criteria. In some cases, we include additional discussion to explain our validation choices.

5.1.1 Effectiveness: Evaluation of Residual Risk

Definition:

When comparing two COAs, our approach should identify the COA that better balances the tensions between the requirements of graceful degradation. For our analyses, it is sufficient to assume that inputs are within a reasonable range.

Measure/Metric:

The residual utility of a system given a specific attacker profile.

Methodology:

Pairwise comparison between small architectures. One architecture alternative will align with secure architecture patterns from trusted sources. The other will be similar but just different enough to not align with best practices.

Criteria:

The architecture alternative that aligns with the best practices should have a higher residual utility as estimated by our approach.

Discussion:

Our pairwise comparisons rely on the ability to objectively determine which of two competing architectures is more secure. However, it can be difficult to make these claims objectively. SMEs may determine that a particular architecture alternative is superior based on experience, but an explanation of *why* that architecture is better might be more difficult to coax out and even more difficult to quantify. Our approach attempts to fill these gaps by offering an *objective and quantifiable* comparison that provides artifacts that can be used to answer the question *why* one architecture alternative is better than another. Since human subjects information is not readily available, we turn to best practices as a way to solve our dilemma about evaluating effectiveness.

We make two key assumptions here. The first is that the best practices for secure system architectures are, indeed, superior practices. Second, we assume that an approach that demonstrates – via small examples – alignment with best practices captures the key aspects of secure architectures and evaluates them in a manner that scales to demonstrate – via larger examples – continued alignment with best practices.

To this end, we have identified a number of best-practice secure architecture patterns that are testable by our approach. For each of the best practices, we provide at least two possible architectures – one that is consistent with the secure architecture pattern in question and one that is not. These architectures are designed to be simple vignettes, so the evaluation has fewer possible confounding factors to skew the result, but in some cases, best practices are used in combination to create a desired effect.

Diversity

Diversity is sometimes also called heterogeneity. The concept is to have components that have different failure modes. For example, rather than having a primary and backup system that are both Microsoft-based, one might choose the primary to be Microsoft-based and the backup to be Linux-based. This can be a double-edged sword, since this adds to the attack surface. Judicious use of diversity can have benefits for security [45].

In our firewall example depicted in Figure 5.1 we show two architecture options. The option at the top shows the attacker is connected to two firewalls of type (i.e., brand) A. Each of the type A firewalls is connected to a type B firewall. The two type B firewalls are connected to a database, which is the component we wish to protect from the attacker.

This example is very small and lends itself to human reasoning with little mathematics. It is clear from the top example that the attacker needs an exploit for a type A firewall and an exploit for a type B firewall to reach the database. At that point, she has two possible paths of equal difficulty. If we wanted to formalize the cost, it would be:

$$c = x_A + x_B + x_{db} \tag{5.1}$$

where x_A is the cost of exploiting a type A firewall, x_B is the cost of exploiting a type B firewall, and x_{db} is the cost of exploiting the database.

Our second example for comparison has the attacker connected to one type A firewall and one type B firewall. The type A firewall is connected to a second type A firewall, which is itself connected to the database. The type B firewall is connected to a second type B firewall, which then connects to the database.

The comparison is clear: In this second example, the attacker needs *either* an exploit for a type A firewall (which she uses twice in a row at no marginal cost) or an exploit for a type B firewall (also used twice in a row). This architecture is less secure than the previous one.[2] The cost for the attacker to reach the database is:

[2] An important caveat is that we are evaluating how security resilience impacts functionality. Other factors such as availability are excluded from our evaluation. From an availability perspective, the second architecture is superior, since it is resilient to a failure of either both type A firewalls or both type B firewalls, whereas the first example architecture is resilient to neither of those cases.

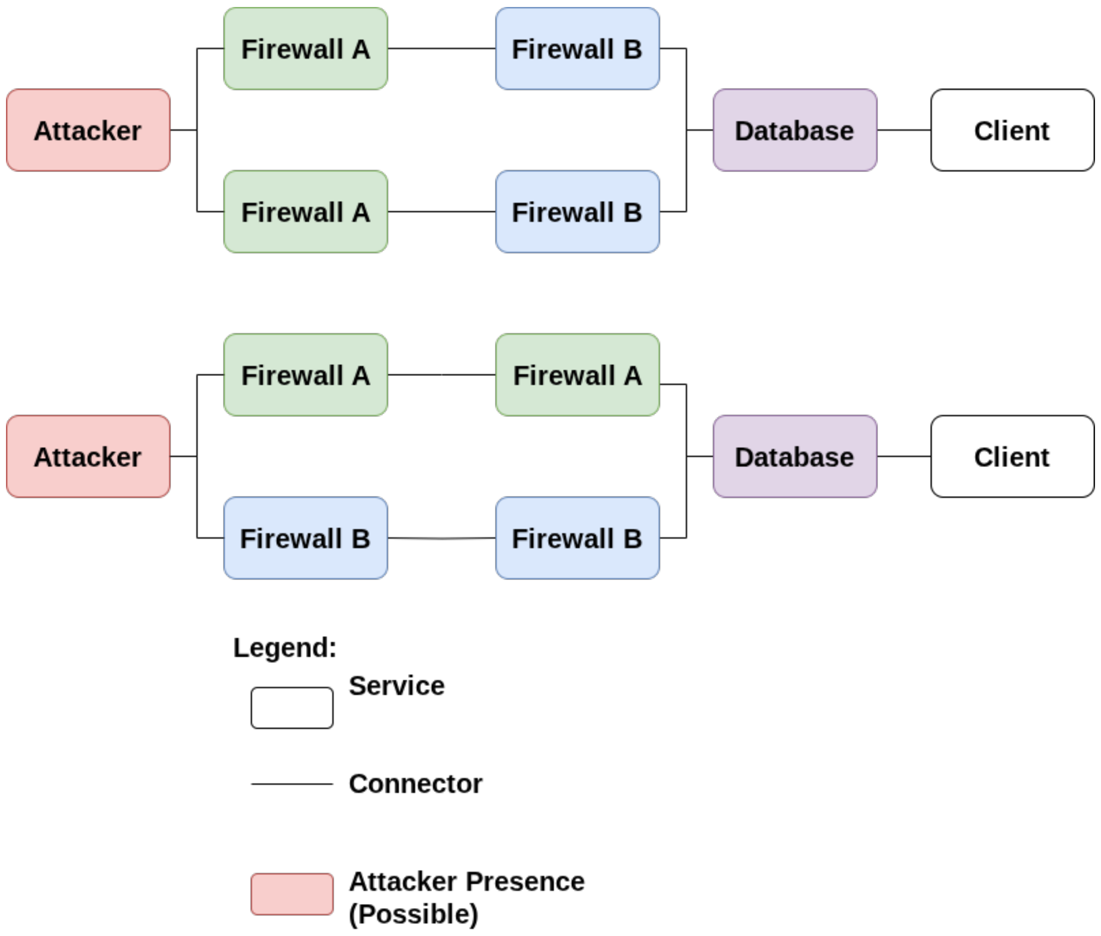

Figure 5.1: Allocation View depicting two possible topologies of the same security components. The top topology is more secure; the bottom is more available.

$$c = \min(x_A + x_{db}, x_B + x_{db}) \tag{5.2}$$

For our evaluation, we make simplifying assumptions that both firewall types cost one unit to exploit, the database costs another unit to exploit, and the attacker's probability of having a particular capability (i.e., budget) is evenly distributed as 20% each for zero, one, two, three, and four units of capability. The utility of the client being able to communicate with the database is 100 utils.

For the first configuration, we have a 20% chance of 100 utils (zero attacker capability), 20% chance of 100 utils (one unit attacker capability), 20% of 100 utils (two units attacker capability), 20% of 0 utils (three units attacker capability) and 20% of 0 utils (four units attacker capability). The expected value is 60 utils.

For the second configuration, we have a 20% chance of 100 utils (zero attacker capability), 20% chance of 100 utils (one unit attacker capability), 20% of 0 utils (two units attacker capability), 20% of 0 utils (three units attacker capability) and 20% of 0 utils (four units attacker capability). The expected value is 40 utils. This configuration is less secure.

The output of our approach is below. It shows that as the attacker capability increases, the affected functionality changes.

Our approach determined that the more secure option is the second configuration, which is intuitive in this small example.

Perimeters and Tiers

In computer networks, the perimeter pattern is that of grouping components of a similar trust level together in a subnet. Subnets have security services between them to mediate communications. Perimeterization is advocated for by the US Department of Homeland Security [46]. Cisco implements this through the concept of "security zones" [47]. Each security zone has an assigned trust score from zero to one hundred. A component like a firewall sits between subnets of different trust levels and mediates the connections. An example initial configuration would allow all connections from higher trust zones to lower trust zones while denying all connections from lower trust zones to higher trust zones. Exceptions are defined in the configuration as needed.

Intuition for perimeterization can be gleaned from research on attack surfaces [39]. Each device in a subnet has its own attack surface. If there is no mediation of access between an external attacker and the subnet, the attack surface of the subnet is the sum of the attack surfaces of each of its components. However, by placing a security device like a firewall between the subnet and the attacker – establishing a *perimeter* around the subnet – the attack surface of the subnet (i.e., the things directly accessible to the attacker) is the sum of the attack surface of the security device and the attack surfaces of the services listening on any of the explicitly allowed connections into the subnet.

Complicated critical systems use a defense strategy that consists of dividing systems into multiple tiers based on considerations of their functionality, criticality, and requirements for trust. The US Department of Homeland Security recommends “Implement a network topology for ICS that has multiple layers, with the most critical communications occurring in the most secure and reliable layer” [46]. Their guidebook for Industrial Control System (ICS), contains a tiered architecture that they recommend as a template for network architects to use [48]. Elsewhere, US federal guidance refers to one use of this practice as “segmentation” or “domain separation” [45]. An alternative to grouping components by security requirements is to group by function [49]. This may help to avoid the loss of multiple functions at once.

Figure 5.2 shows several different configurations for a network. In each, there is an attacker component representing an untrusted internet, a corporate workstation, a corporate printer, a server with financial data, and a server with inventory data. For the important function of finances, the financial server consumes inventory data from the inventory server. For the slightly less important function of inventory management, the corporate workstation consumes the same inventory data. The third, far less important function, is printing, in which the printer receives print jobs from the workstation. While the internet is definitely compromised, there is a 50% chance that the printer is.

Architecture A is in line with best practices. The workstation and printer are on one subnetwork, while the financial and inventory servers are on another. Firewalls separate the internet from the first subnet, and also the two subnets from one another. Our evaluation provides this with alternative with the highest utility (40.0 out of 100.0).

When there are two points of compromise, there is a benefit when those points of compromise are separated from each other and from other network functionality. The addition of the “insider threat” justifies the tiered architecture. With a possible attack originating from the printer, our critical back-end functionality between the two servers must be protected from both the printer and the internet-based attacker components.

Architecture B also uses a tiered approach. However, this example makes a disastrous decision to locate the inventory server – needed for the two highest utility functions – adjacent to the possibly-compromised printer. Our evaluation yields a utility of 29.0.

It is key that the subnets in our tiers attempt to isolate functions (based on their consuming and producing components) from the points of compromise. If one subnet contains components critical to many functions, and that subnet also contains a compromised components, the number of functions that the attacker can compromise increases substantially compared to a best practices approach in which functions are more isolated from each other.

Architecture C creates a single, strong perimeter. The two firewall components are placed back-to-back to separate the internet from the internal network. Because the printer is com-

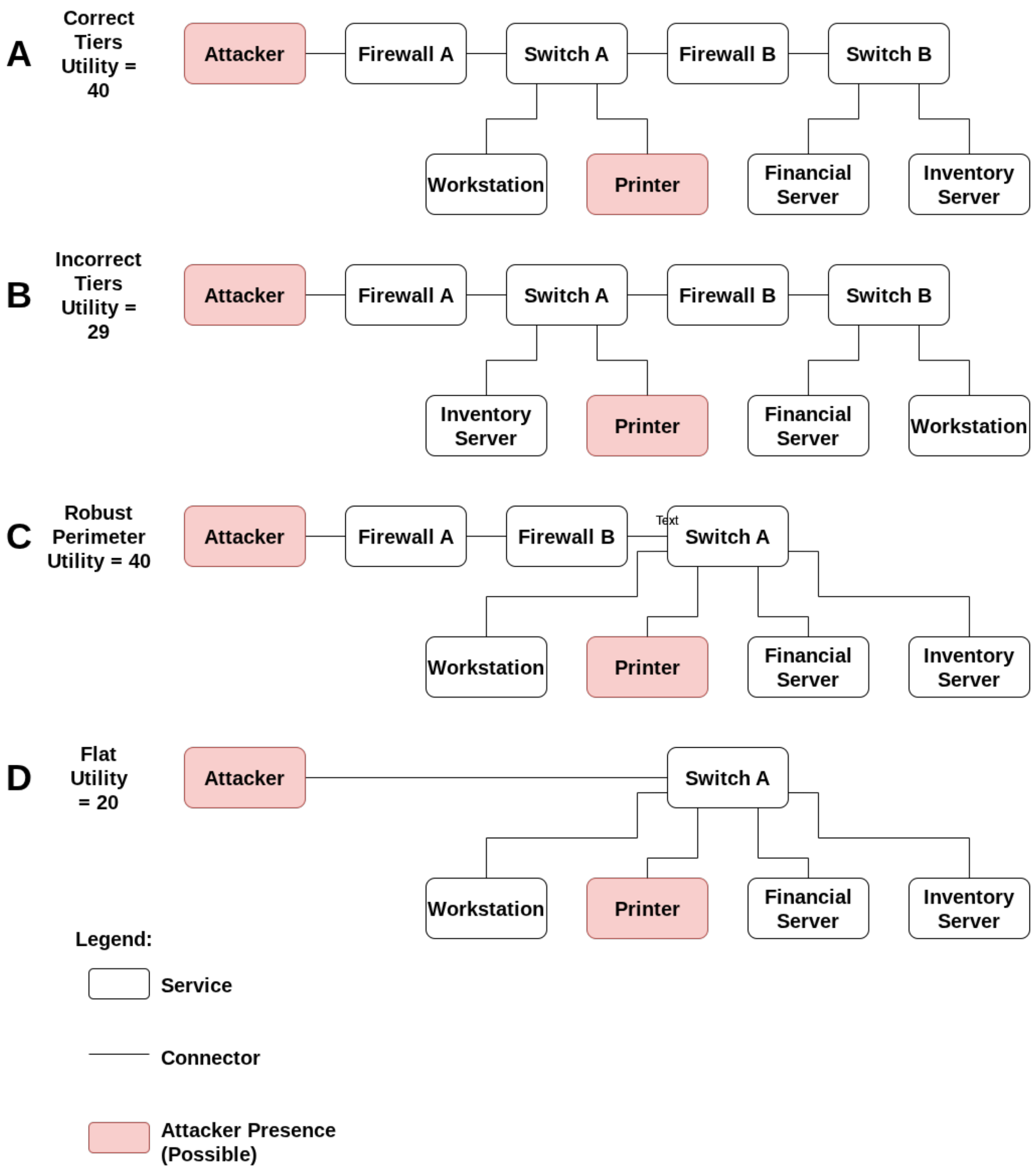

Figure 5.2: Deployment View of networks with different defensive architectures.

promised 50% of the time, this architecture has the same utility of 40.0 as the correctly tiered architecture (Architecture A). It balances the advantages of bolstering the internet-connected part of the network with the disadvantage of having no separation on the internal network. If the printer were less than 50% likely to be compromised, our approach would prefer the correctly tiered architecture.

Architecture D shows a flat network with no perimeters. As we would expect, this network has the lowest utility; our evaluation yields a utility of only 20.0.

We also ran our analysis on several variations of the exemplar system from Section 3.5. In the first set, we moved the printer — a low value component. With the printer on the high trust control network, the residual utility was 68. With the printer connected to the lower trust network (SwitchB), it was 149 — a best practice alternative. In the second set, we kept the printer on the low trust network and moved the SCADA server — a high value component — to the same low trust network, dropping the utility to 67. The alternative with the highest residual utility aligns with current best practices for ICS system architectures [50].

Redundancy

When a component fails, it can be helpful if another component is available to provide the necessary functionality. Of course, this requires the proper placement of the component, since if these components all are collocated in the network topology and have the same failure mode, they may all fail at once. This pattern may be implemented with the Diversity pattern. Redundancy is called out as a security strategy in guidance from the US [45].

We test redundancy with the following vignettes:

- This architecture alternative has a collocated backup component of the same type (i.e., with no diversity) for a critical component.
- This alternative has redundancy that is collocated and diverse.

Evaluation:

The two architectures are shown in Figure 5.3. Both have two components that can perform the server functions. The top architecture has a server and backup server of different types. This architecture degrades well because the attacker must be compromise both servers (i.e., using two different exploits) to cause a loss of the server-related functionality. Our tool evaluates this architecture as having a utility of 60.0.

The bottom architecture contains a server and backup that are of the same type, so they have the same vulnerabilities. Because of this, both the server and its backup have the same failure mode, and we assume the attacker compromises both because it does not cost her more to do so. This backup strategy does not provide a security benefit, so our tool evaluates this architecture as having a utility of 40.0.

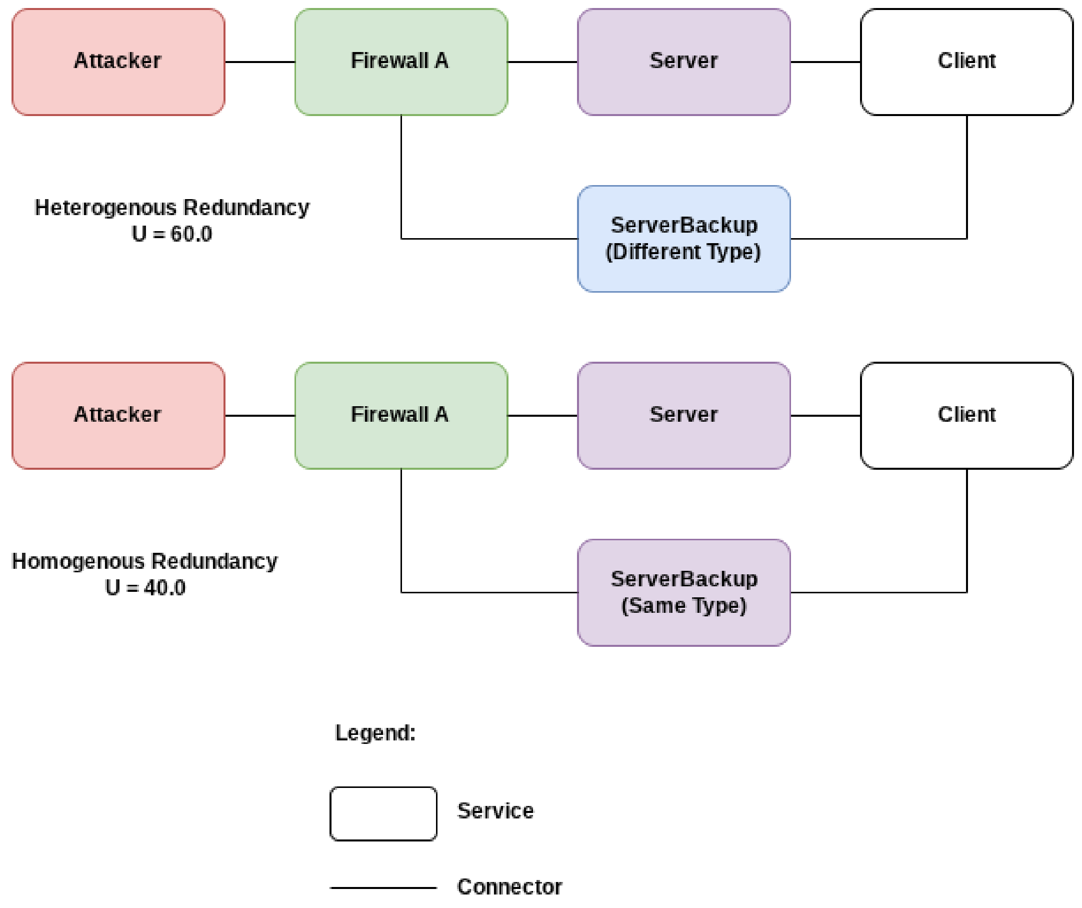

Figure 5.3: Deployment View of a network with primary and backup components.

Least Privilege and Least Sharing

Least privilege and least sharing are overlapping concepts that encourage the assignment of minimal functionality to each system component [45]. That way, when a component fails, the impact is limited to a smaller set of functions. Least privilege is an excellent principle; our approach can provide more nuance. Additionally, those functions must contribute to higher level functions. If one function (e.g., failure of an electrical relay) is going to turn off the power for a neighborhood, another failure of a different function leading to the same loss does may reduce the utility.

This pattern is related to the Perimeter and Tier patterns. In this case, perimeters are applied around components of similar functionality; this is done is a way that minimizes the number of functions depending on the security each subnet.

We test least privilege / least sharing with the following vignette:

- One of our architecture alternatives has a subnetwork that supports multiple functions.
- The other alternative separates the components across subnetworks in a way that minimizes the need for support multiple functions in a single subnet.

Evaluation:

We validate this concept in our earlier evaluation of perimeters and tiers. In Figure 5.2, the second architecture allocates components to subnets in a way that causes unnecessary overlap of functional dependencies on each subnet. The improper allocation results in a suboptimal architecture when compared with the top architecture in the figure.

5.1.2 Scalability: Evaluation of Architecture Alternatives

Definition:

Scalability is the manner in which the computing and/or temporal resource requirements of our approach increase with respect to the size and complexity of the input system models.

Measure/Metric:

The number of input components, complexity of input components, and a polynomial or exponential expression describing resource-consuming actions like creating an architecture alternative or generating or querying a set of attack traces.

Methodology:

Manual evaluation of the input model, utility calculation, and COA search. Verification through experimentation.

Criteria:

Representative models from respected, open sources should be processed for COA identification in eight hours for real time evaluation and five days for design time evaluation.

Evaluation:

In the current implementation of our approach, we bound the generation of architecture alternatives by requiring them to be within a pre-specified graph edit-like distance from the original architecture under consideration (see Chapter 3 for more details). Restrictions on scaling come from two primary sources: 1) the number of architecture alternatives that are possible and 2) the complexity of analysis of each architecture alternative.

In the architecture and design phase of the system development lifecycle, there is no cost to change architectures because none have been implemented. While this may increase the number of plausible alternatives, there is no need for near real time outputs from our approach. For the purposes of scalability, we currently require the seeding of our algorithm with architecture alternatives, and we perform the graph edit distance bounded evaluations from these seeded architectures. However, other implementations could be more efficient while conserving the basic concepts of our approach.

At run time, the need for speed and scalability is more important because an attack may be imminent or ongoing. Our current implementation performs an exhaustive search through all the possible sets of defensive tactics given a particular defender budget as applied to the initial architecture. This is an anytime algorithm implementation, so we could output the best adaptation recommendation found so far. Another option is to precompute various attack scenarios so they can be rapidly applied if and when a similar attack occurs.

Scaling richness (i.e., realism expressed through modeling additional system and environmental details) and number of components and connectors has a direct correlation on the time to evaluate the expected utility of an architecture post-COA implementation. It also has a direct correlation on the time to search across the COAs to find the optimal COA and corresponding architecture alternative. As a graph, the number of possible edges grows with the square of the number components. This corresponds to the total number of possible architecture alternatives. For each architecture in a fully connected graph, the number of possible paths (for attacker movement or data flows) through a graph grows on the order of $n!$, where n is the number of nodes. And when we generate attack scenarios, if there are a number of attackers, there are 2^a possible combinations of attackers. These all contribute to the growth of state space in our approach.

For one of our evaluations of the ability to scale, we ran both the security and availability configurations of our firewall example three times each on three different configurations of instances of Github Codespaces. The three different configurations were:

- Small: 2-core, 8GB RAM, 32GB storage
- Medium: 4-core, 16GB RAM, 32GB storage
- Large: 8-core, 32GB RAM, 64GB storage

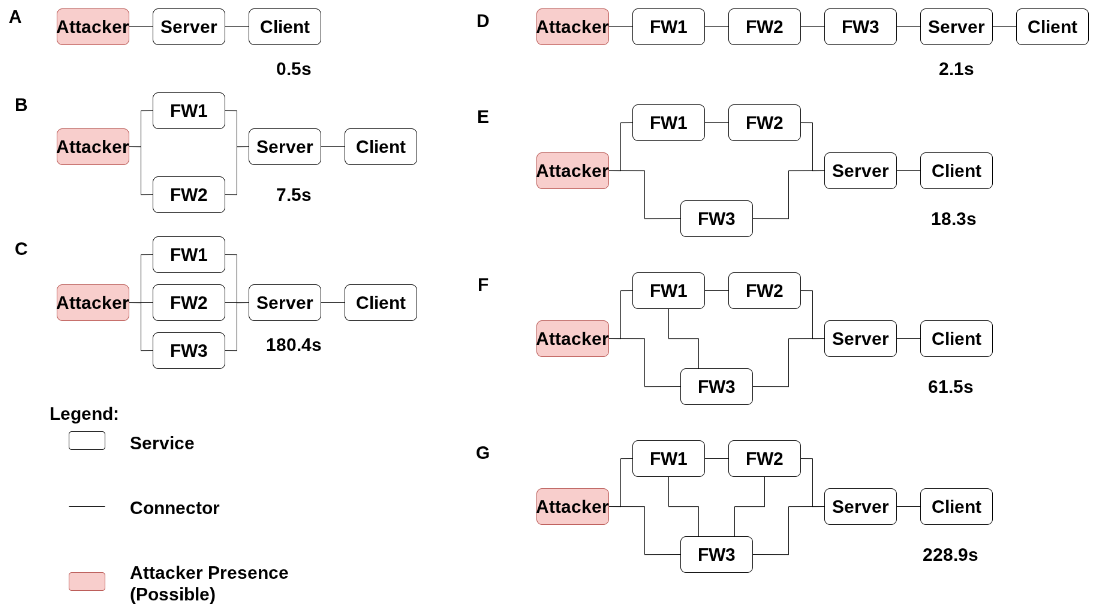

Figure 5.4: Architecture size and connectivity impact on evaluation times.

We found that the time to evaluated the security configuration (see Table 5.4) and availability configuration (see Table 5.5) did not change as the machine type grew more capable. We believe this is because pyDatalog runs in a single thread, so it does not benefit from additional CPU cores. Some of the work could be parallelizable (e.g., determining the possible data flow paths and computing the possible attack scenarios), so there are opportunities to make the evaluations more efficient.

We also evaluated scalability in terms of the number of nodes, edges, and branches in the Deployment view of our architectures. The architectures and times to evaluate are shown in Figure 5.4. Increasing the number of either nodes or edges increases the evaluation time. In particular, when the graph becomes more dense, evaluation times increase to reflect the possible branches that could be taken by the attacker and the data flows.

Architectures A, B, C in the first column demonstrate the impact to performance when we add branches, each with an additional component. In the second column, architectures D, E, F, and G each have the same number of total components, but the number of interconnections increases from one architecture to the next. As the architecture becomes more dense from interconnections, performance decreases – even if the number of components remains constant.

5.1.3 Uncertainty in Tool Inputs: Sensitivity

Definition:

Sensitivity is a measure of the variation of outputs with respect to the variation of the inputs. This is to ensure SME inputs do not have to be precise to be effective.

Measure/Metric:

Percentage change in value of an input parameter; percentage change in residual utility; COAs selection.

Methodology:

Pairwise comparison of input to output selecting inputs evenly spaced from zero to one hundred percent input change in no greater than 1% increments.

Criteria:

Oscillations of outputs should not appear across input changes.

Evaluation:

We have evaluated this approach on the exemplar system from our running example described in Section 3.5. In our evaluation, we assessed the output results (i.e., were the mitigations sensible), the sensitivity of the outputs to changes in the inputs, and the performance of the evaluations.

The results matched best practices. We were able to demonstrate that a misplaced component – the printer on the control system network – posed a higher risk to system resilience than connecting the printer to a switch on the internet-side of the firewall. By deliberately misplacing the switch, we were able to generate a COA to move the printer to this better location.

A major concern was the sensitivity of our approach to subjectively-derived quantitative inputs. An example of this is the probability of compromise of a particular component. For example, as the probability of compromise of the Virtual Private Network (VPN) is increased from 80 to 81 to 82% and so on, does the estimated residual utility of the system vary widely, or does it change slightly? We demonstrated that the utility changed smoothly, in small quantities. This suggests that, as long as the subjectively-derived quantitative inputs are in the "ballpark," the evaluation outcomes should be sensible.

We evaluated the sensitivity of our approach to changes in attacker capability, probability of component compromise, and changes in utility. These variables are derived from expert knowledge, and experts can reasonably disagree about the specific values, so our goal was to demonstrate that minor disagreements in input values do not result in major differences in evaluation outputs.

The primary mission of this system is electrical transmission, so we assigned this top-level functionality a high utility of 100. The transmission can run autonomously for a short period, but it ultimately needs management to continue, so we chose a utility of 50 for transmission management. Logging is important for billing and compliance purposes, so we assigned it a utility of 20. The enterprise computer systems are not critical to the main purpose of the system. The ability to work from home is similarly helpful but not critical, so we assigned each a utility of 5. Thus, the total utility of our system is 180.

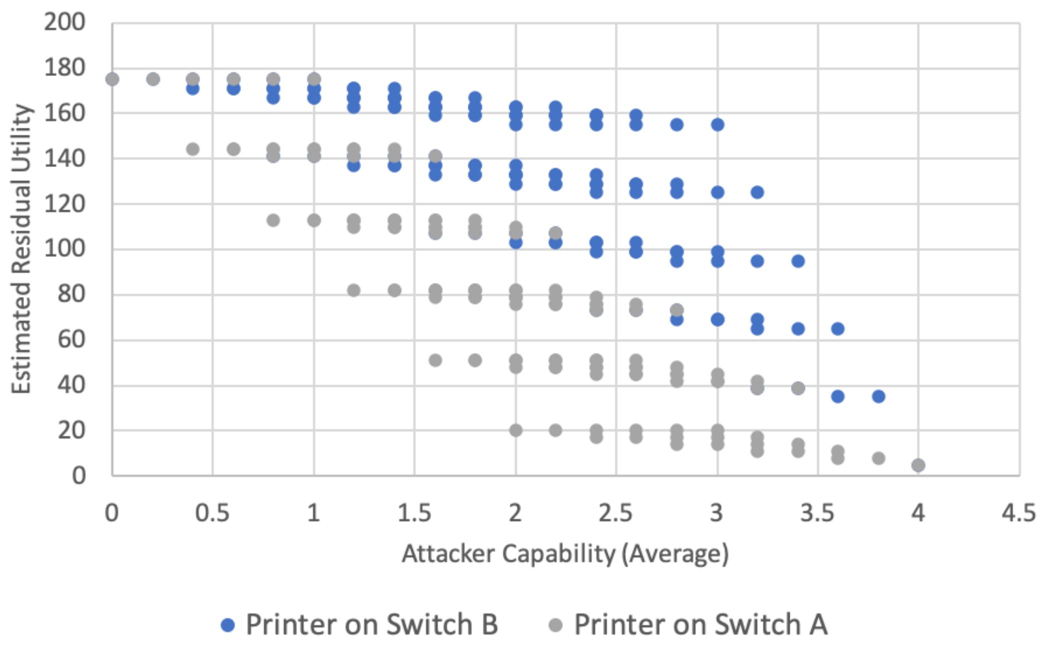

Figure 5.5: Sensitivity of Evaluation to Attacker Knowledge.

For evaluation purposes, the attacker capability estimate is from zero to four exploits, with a probability of 20% for each capability.

To begin with, our system's initial estimated residual utility is 165.4. This is because we specify that the VPN is compromised with a probability of 90% and the printer with a probability of 10%. Note that both or neither devices can be compromised because their probabilities of compromise are independent.

Attacker Capability

To gain a better sense of the impact of attacker capability distributions on utility, we evaluated an attacker with a capability range of zero to four. For each distinct capability value, the probability the attacker has that capability could be 0, 20, 40, 60, 80, or 100%. The sum of the probabilities of the different possible capabilities must be 100%. Thus, we explored the range of capabilities, including Black Swan events in which a low probability, high impact event might occur. The results are depicted below in Figure 5.5. As can be seen, the residual utility is sensitive to the degree of uncertainty of attacker capability. It gradually slopes downward as attacker capability increases. The step-wise decreases in utility are a result of the all-or-nothing aspects of the way functionality utility contributes to total utility: If a functionality is considered compromised, its fully utility is lost. Most average attacker capability values have multiple gray (or blue) dots organized vertically because there are multiple possible capability probability distribution functions with that specific mean capability value.

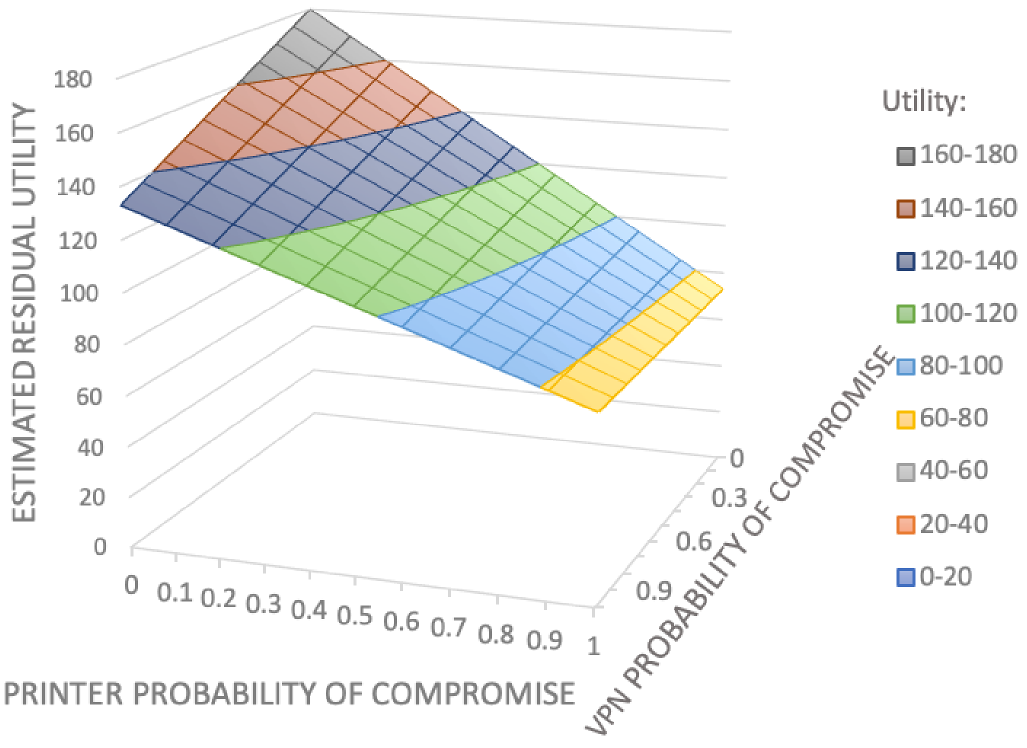

Figure 5.6: Sensitivity of Utility to Probability of Component Compromise.

Probability of Compromise

The probability that a given component is compromised affects utility. In our analysis, we consider the probabilities of compromise for the printer on the control network in ranges from 0-100% (increments of 10%) and compromise of the internet in ranges from 0-100% (increments of 10%). The probability of compromise of each of the components is independent. The results are depicted in Figure 5.6 below. For this analysis, we assumed the probability of any given attacker capability was evenly distributed from zero to four, and the printer was co-located with the control system section of the network.

Note that the printer's probability of compromise has a more dramatic impact on residual utility than the VPN's probability of compromise. This is what we would expect because of the printer's proximity to high-utility functionality.

Utilities

Consider a scenario in which either the transmission or enterprise utilities are increased by 15. Changing the enterprise utility from 5 to 20 changes the residual estimated utility from 149 to 163. Changing the transmission utility from 100 to 115 also changes the residual estimated utility to 163. The implies that, if utilities of functionalities are selected in a manner that is roughly proportional to each other, the outcome of the evaluation should be reasonable.

5.1.4 Usability: Labor Requirements

Definition:

For labor usability, we consider the inputs necessary to implement our approach. In particular, we consider the labor to produce or format these inputs – labor that would not be required but for our approach. This additional labor can include using existing skills, acquiring new ones such as Datalog, or involving personnel that are not traditionally part of the system life cycle.

Measure/Metric:

We expect that architecture SMEs with Datalog experience will create the architecture styles for systems. Threat SMEs with minimal knowledge of Datalog will provide inputs like attacker descriptions. Threat SMEs are not traditionally part of system development life cycles, though it is not uncommon for threat briefings to be provided by threat SMEs to system stakeholders for critical infrastructure and national defense systems. Architecture views are defined either by architects or system administrators. SME labor needs can be estimated by the number of parameters, attributes, and architecture style rules that must be input by SMEs. The number of SME-input parameters, attributes, and rules that change over the course of design time, run time, or complete system lifecycle. These are grouped by type of SME.

Methodology:

Manual evaluation to identify what must be input by a SME, scaled to the size of our evaluation systems.

Criteria:

Inputs to our approach should not require hiring new personnel for specialized skill sets. Additionally, the architecture views should not require significant Datalog experience; they should also use existing architecture artifacts where possible and not require major amounts of effort to produce or format. Architecture style rules should not require updating.

Evaluation:

Requirements for additional labor – particularly for new types of labor – are minimal with our approach. The Deployment View is a common architecture view that we expect to be readily available for any well-built system. Each component is described by a few lines of Datalog (e.g., one to define the type of component, one to define if it holds any credentials, one to define if it uses any credentials, one line for each network connection, etc.) A small number of lines of code construct the hierarchy of component types (e.g., Windows 11 specific version, Windows 11, Windows). At architecture and design time, we expect the architect will anticipate provide a rough idea of the type (i.e., Windows rather than a specific version). At run time, automated network discovery tooling can identify components on a network, aiding the production of the

Deployment View. The implementation of this view is aided by following the templates of the smaller examples in this paper.

The most novel part of the Deployment View compared to a traditional Deployment View is the need to define levels of vulnerability of (i.e., expected cost to exploit) the various component types. A naïve implementation can just use a value of 1.0. Our small examples used exactly that. As architects or administrators gain new information (e.g., through examination of relative costs for exploits for sale on the black market), these values can be adjusted, and the evaluations should be rerun with the new values.

The Component-and-Connector (C&C) View requires more knowledge, but existing commercial tooling like net flow analysis tools identify data flows that an architect or system administrator can label and format for our approach. This requires one line of Datalog code to represent each data flow producer (source) and one line to represent each data flow consumer (destination). An architect or administrator should know the consumers' needs for confidentiality, integrity, and availability. The implementation of this view is aided by following a template like one of smaller examples in this paper.

The Functional View is the least amenable to automation and also may not be readily available. Unlike the Deployment View and C&C View, system functions are not discoverable via automation. These functions must be described by a well-informed architect or system administrator. At architecture and design time, it is reasonable to expect that the architect will understand the purpose of each of the components in the system. At run time, the starting point for the Functional View is the C&C View, since the data flows must be associated with system functions. System stakeholders will need to work together to produce the Functional View by assigning data flows to functionality and producing the hierarchical tree of functional requirements.

Each function is described with one line of Datalog code. The tree-like hierarchy requires additional lines of Datalog code, but this code is structured simply and can be implemented by following a template. Most of the Functional View will stay static over the system lifecycle, but some changes to data flows may cause minor changes of the leaf nodes in this view.

While architecture views are the most straightforward to create, architecture styles (i.e., constraints on architectures based on the type of system) require careful creation by a SME who is familiar with both Datalog and the type of system for which the style is being designed. The styles take the form of a series of rules that constrain how components interact. As an example, one could define that connections are bidirectional. These rules are defined once per system type, and they can be reused for systems of the same type. In our examples, if the system type did not change, then changes to the styles were unnecessary.

The table below summarizes the need for creating wholly new artifacts, developing new skills, the ability to automate data collection and formatting, and whether or not the data changes over the course of the system life cycle.

Requirement	New Artifacts	New Skills	Automatable	Mutable
Deployment View	No	No	Yes	Yes
C&C View	Maybe	No	Yes	Yes
Functional View	Likely	No	No	Limited
Architecture Styles	Yes	Yes	No	No
Attacker Capability	Yes	Yes	No	Yes
Attacker Points of Presence	No	No	Yes	Yes
Defender Budget	No	Yes	No	Yes

5.1.5 Usability: Explainability

Definition:

Explainability is the ability of the artifacts produced by our system to be able to defend the recommended architectures. This does not necessarily mean that a human is convinced, but it may help a human understand and agree with the output of our system.

Measure/Metric:

Are artifacts sufficient to understand counterfactuals?

Methodology:

Demonstration that artifacts can explain why one output architecture is more secure than another.

Measure/Metric:

The optional and available artifacts that can be provided by the approach.

Criteria:

Key information to show the inferiority of an alternative architecture should be present.

Evaluation:

We use the following pyDatalog query on individual architecture alternatives in our evaluation:

```
query = "(worstCaseScenarioByStart[CompromiseSet,PC,TotalC] == X) & (
    estResidualU[True] == U)"
```

The output provides a number of useful artifacts that can aid and explain:

- The estimated utility for the architecture after accounting for the possible attack scenarios.
- For each possible combination of attacker starting points:
  - The probability of that combination of attacker starting points.
  - For each possible attacker capability level:
    - The exact sequence of steps in the attacker scenario, including which exploit was used and on which component.

  * The estimated best path for each data flow, if applicable, and its resulting ability to guarantee confidentiality, integrity, and availability.

This information explains, for each of the possible combinations of attacker starting points and attacker capabilities, the estimated worst case attack scenario step-by-step and its ultimate impacts to the data flows of the system being defended.

A sample of output from the evaluation of the Homogeneous Redundancy architecture in Figure 5.3 is shown here:

```
(   ('attacker',),
    1.0,
    2.0,
    (   (   ('attacker', 'attacker', 'compromisedattacker'),
            ('attacker', 'fwA1', 'fwAExploit'),
            ('fwA1', 'server', 'serverExploit'),
            ('fwA1', 'serverBackup', 'serverExploit')),
        (   (   (   'userAuthorization',
                    ('serverBackup', 'fwA1', 'server', 'client'),
                    0.0),
                'databases',
                (),
                0.0),),
        0.0),
    40.0),
```

The attacker component is the one that is compromised. The chance of this happening is 100%. The attacker has a capability of 2. The attacker's steps are first the compromise of itself, then the generation of a fwAExploit to compromise fwA1. Then a serverExploit is used to compromise the server. The serverExploit is reused to compromise the serverBackup. This causes the data flow to be compromised such that no functionality is provided at greater than 0 utils. The total utility of this architecture is zero under this attack scenario. Overall, the architecture has an estimated utility of 40 utils when evaluated across the entire range of the attacker profile.

5.1.6 Realism: Attacker Richness

Definition:

Attacker richness is the extent to which our model captures the salient details of real world attacks.

Measure/Metric:

Evaluation of the usefulness of attacker and attack attributes not in our model.

Methodology:

Manual evaluation to identify key attributes of industry best practice models for attacks. Comparison with the attributes used in our models.

Criteria:

Attributes in our models should cover the key real world attributes identified.

Evaluation:

The attacker is an external actor (with respect to our system and its stakeholders) about which we deliberately assume as little as possible. We do have to make some assumptions. For example, knowing the possible starting points of attacks is key, as we have seen earlier in this chapter, to ensuring the proper placement of defenses such as perimeters and tiers. The probability of an attacker having access to those possible starting points is key to determining how likely a defense is to be useful when placed in specific areas of the network topology. The capability of the attacker is absolutely essential – without this, it is impossible to know if defenses are insufficient or overkill.

We do mix attacker points of presence and attacker capability ranges. For example, we do not currently have a way to describe that an attacker from the internet is more likely to have higher or lower capability than an insider. Instead, we consider one attacker at a time, and that attacker encompasses all likely points of presence and capabilities. This keeps the modeling simpler and avoids further state space explosion issues. Based on our prior evaluations, this sacrifice does not degrade the outputs in any significant way.

The US defense contractor Lockheed Martin developed a “cyber kill chain” that describes the steps in an attack [51]. A US Department of Defense research and development center MITRE developed a competing taxonomy called ATT&CK [52]. ATT&CK is more detailed than the cyber kill chain, but the two overlap significantly.

In each, the first step is reconnaissance. This can be performed online, offline, or via a mix of both. We do not represent reconnaissance. This is because it can be performed offline – and therefore out of scope of our approach – and also because we assume a worst case scenario omniscient attacker.

The next step is weaponization (Cyber Kill Chain) or resource development (ATT&CK). This could include the actual creation of an exploit. We represent the cost to create an exploit in the estimated level of vulnerability of component types and in the limited budget of the attacker.

After that, there is delivery (Cyber Kill Chain) or initial access (ATT&CK). There are multiple ways this could occur. We have shown how we model an adversary making incremental progress by beginning their attack on the internet and working their way through defenses one at a time. Initial access deeper in a system, such as through phishing, is slightly different, since the attacker point of presence may seem to originate from the phishing-compromised component, and it can be modeled that way.

The next two steps are exploitation and installation (Cyber Kill Chain), or execution and establishing persistence (ATT&CK). We represent exploitation/execution in attack scenarios each time an exploit is applied. While an attacker may not establish persistence on a component that she exploits, we assume the worst case scenario – that she establishes a continued presence on each exploited component.

ATT&CK continues with a series of tactics not explicitly enumerated by the Cyber Kill Chain. These are Privilege Escalation, Defense Evasion, Credential Access, Discovery, Lateral Movement, and Collection. While we did not evaluate for privilege escalation in our validations in Chapter 5, this step can be represented by dividing a host into user-level and superuser components, with style rules governing movement between the two. Defense Evasion is an observability concern, which is out of scope of our approach. Our approach represents credentials (e.g., passwords and session tokens) as items that can be stolen from specified components to be reused on others. Discovery is similar to reconnaissance; here again we assume the worst case scenario of an omniscient attacker. Attack scenarios show lateral movement with step-by-step artifacts enumerating the sequence of components compromised by the attacker. Collection is when the attacker gains access to the data types that she wishes to compromise. In our approach, a data flow is considered compromised when an attacker compromises any component that produces, transmits, or consumes a data flow.

The next common step for both the Cyber Kill Chain and ATT&CK is Command and Control. In this step, the attacker maintains communications with exploited components to continue attack objectives. We assume a worst case scenario – that the attacker has unfettered access to components once compromised – though a future extension of our work could add nuance to demonstrate how actions to restrict attacker command and control can benefit defenses. However, with the continued advancement of machine learning, semi-autonomous or fully autonomous malware could reduce the need for command and control.

Finally, we have Actions on Objectives (Cyber Kill Chain) or Impact (ATT&CK). Here, the attacker attempts to achieve whatever her goals are (e.g., data exfiltration, data corruption, system destruction). We assume any compromise of component confidentiality, integrity, or availability can and does achieve the loss of that security attribute. Because of our assumption that the defender does not know the goals of the attacker in advance, we assume the worst case scenario for each component compromise.

5.1.7 Realism: Defender Richness

Definition:

Defender richness is the extent to which our model captures the salient attributes of real-world defenses against attacks.

Measure/Metric:

Evaluation of the usefulness of defender and system under test attributes not in our model.

Methodology:

Manual evaluation to identify key attributes of industry best practices for modeling defenders. Comparison with the attributes used in our models.

Criteria:

Attributes in our models should cover the key real world defender attributes identified that may meaningfully contribute to an evaluation of the secure graceful degradation of a system.

Evaluation:

While the Cyber Kill Chain does not directly address defenses, MITRE maintains a taxonomy called D3FEND [53]. This taxonomy consists of the following categories: Model, Harden, Detect, Isolate, Deceive, Evict, Restore. Deception is outside the scope of our approach. We will discuss how the other categories apply to our approach.

First, modeling is exactly what we do in our approach when we create models for evaluating secure graceful degradation. However, the modeling itself is not a defensive tactic. It is an approach to understanding a system and its properties.

Hardening is the act of making a system more robust against attack. In the D3FEND context, hardening generally is not system-level but rather component-level. Our models can reflect hardening through changes to the model. These changes include raining the expected cost to exploit a component type (e.g., following in-depth static source code review and dynamic application security analysis). For credentials, the name of the credential can be changed to reflect that a password was changed or session token reissued.

Measures to increase detection are outside the scope of our approach. However, if an attacker is detected or suspected, the model should be updated to reflect the known probabilities of presence, points of presence, and capability of the attacker. The defender should rerun the evaluation with the updated model.

Our evaluations are particularly well-suited to evaluating isolation. Isolation can be achieved through removing credentials from a subsystem that should be isolated from the rest of the system, and it can be achieved by removing or reducing connectivity between a subsystem and the rest of the system. The defender can reduce connectivity by the appropriate placement of a security component like a firewall.

Eviction is the act of removing an attacker's presence and access. We model the removal of an attacker's presence by updating the attacker's probabilities of presence and points of presence. Other changes to the model are associated with eviction. For example, one can shut down a system. In our model, that is achievable through removing a component from the system or equivalently through completely isolating that component from the rest of the system. If our defender changes or removes credentials from a component, this act can also be part of the eviction process. If the defender reboots a system to remove malware that is solely based in ephemeral memory, we model this by only updating the attacker's corresponding probability and point of presence.

The final category is restoration. Restoration is the act of regaining lost functionality. We can model aspects of restoration through changes to credentials (including adding credentials back to a component) and restoring connectivity to a component.

5.2 Validation of Non-Enterprise Network Systems

In the prior sections of this chapter, we demonstrated that our approach is applicable to enterprise network architectures and industrial control system architectures (i.e., where tiers are critical to graceful degradation). Our approach is generalizable to other types of systems. In this section, we evaluate our approach with one such system that is not an enterprise network.

To demonstrate the broader applicability of our approach, we simplified and adapted a small satellite architecture [54]. The satellite's communications subsystem connects to a ground control system. The communications then connect to a Controller Area Network (CAN) bus. The CAN connects to two payloads and the navigation, propulsion, and power subsystems. For simplification, we bundle the navigation, propulsion, and power into one component representing the three separate subsystems.

We further have defined two possible points of presence for an attacker. She could be present on the ground control system with a probability of 25%. Additionally, she could have compromised the Payload2 component with a probability of 75%. The two probabilities are independent of each other. The compromise of the payload is the type of concern a satellite operator might have if the payloads are developed by customers or international partners for whom trust is limited.

The utility from the operation of the satellite (not accounting for the payloads) is 70. Payload1's science mission has a utility of 20, and Payload2's science mission has a utility of 10. For our evaluation, we consider how alternative placements of a security device like a firewall promote the ability to gracefully degrade in response to compromise.

The initial architectures is shown in Figure 5.7; with no security device, it has an estimated residual utility of 10.125. The same utility applied when the device was place either between the ground station and communications (Figure 5.8), and when it was placed between the CAN and navigation / propulsion / power component (Figure 5.9). However, when the security device was placed between the CAN and Payload2 (Figure 5.10), the utility increased to 20.25. This is because the ground station is critical to all utility; any compromise of it completely eliminates utility. If Payload2 is compromised, there is some utility possible from the satellite continuing

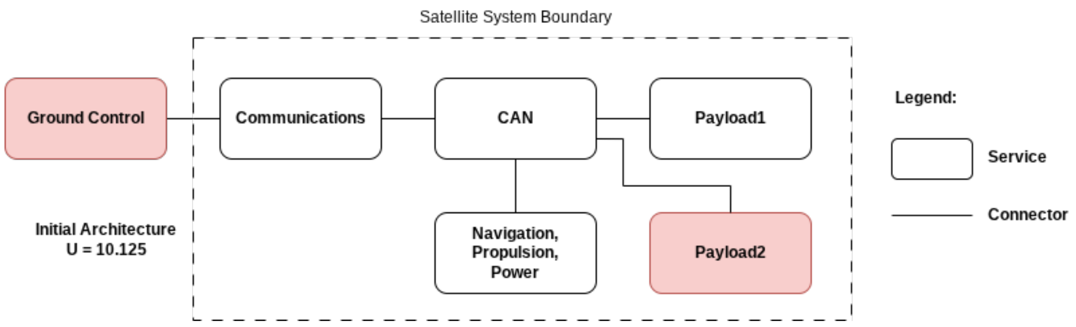

Figure 5.7: Deployment View of initial satellite architecture.

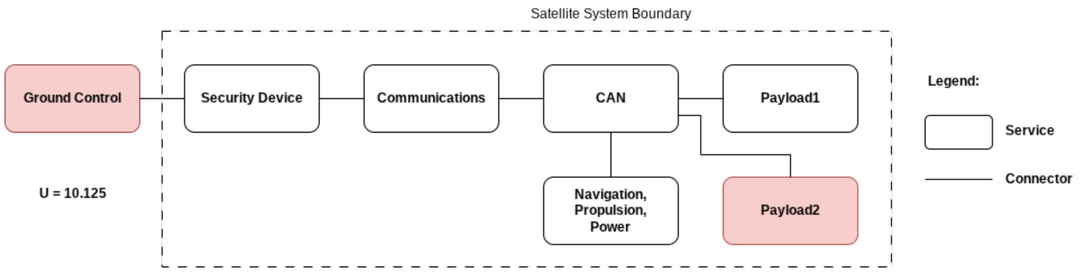

Figure 5.8: Deployment View of initial satellite architecture with protection near ground control.

to operate, as well as a possible utility from Payload1. The evaluations took between 100 and 152 seconds per alternative.

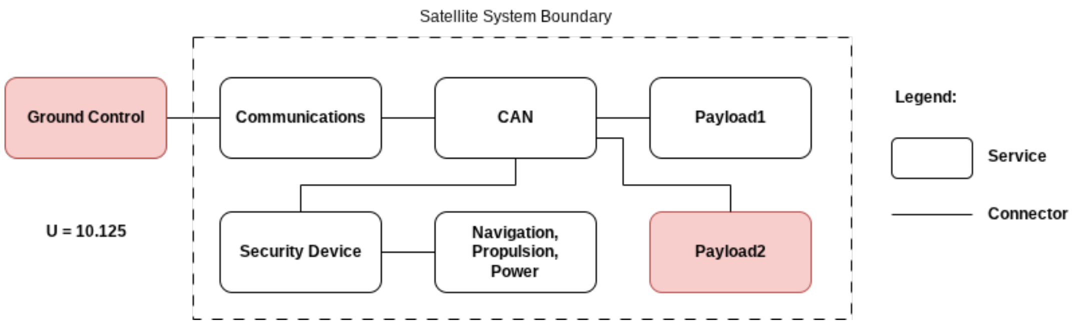

Figure 5.9: Deployment View of initial satellite architecture with protection near navigation, propulsion, and power.

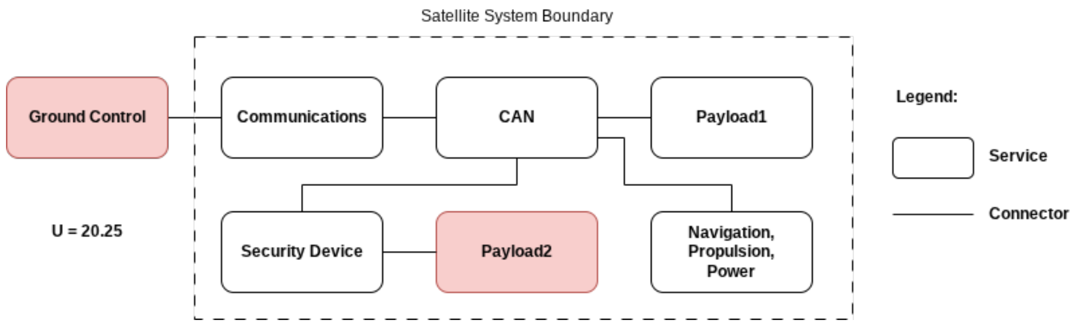

Figure 5.10: Deployment View of initial satellite architecture with protection near Payload2.

Claim	**Requirement**	**Validation Strategy**
Effectiveness	Correctly shows functional effects of attack scenarios	Arguing and Case Studies
	Outputs are reasonable degradations	Arguing and Case Studies
	Works effectively at design time	Case Studies
	Works effectively at run time	Case Studies
Scalability	Scales enough to inform architecture decisions	Case Studies
	Runs in time to be effective at design time	Arguing and Case Studies
	Runs in time to be effective at run time	Arguing and Case Studies
Uncertainty	Sensitivity is low to small differences in inputs	Arguing and Sensitivity Analysis
	Design time uncertainty is appropriately accounted for	Arguing
	Run time uncertainty is appropriately accounted for	Arguing
Usability	Approach is automated	Arguing
	Minimal changes are necessary between design and run times	Arguing and Case Studies
	Inputs are similar to what industry already uses or has available	Arguing
	Compatible with Agile development	Arguing
	SME labor is minimized	Arguing
	Outputs are explainable	Case Studies
Realism	Incorporates multiple views	Arguing and Case Studies
	Correctly shows how attackers move	Arguing and Case Studies
	Shows the types of defenses defenders actually use	Case Studies

Table 5.1: Correspondence between claims and validation approach.

Probability	Capability	Utility
20%	0	100
20%	1	100
20%	2	100
20%	3	0
20%	4	0
Expected Value:		60

Table 5.2: A secure firewall configuration has higher expected residual utility than the available configuration.

Probability	Capability	Utility
20%	0	100
20%	1	100
20%	2	0
20%	3	0
20%	4	0
Expected Value:		40

Table 5.3: An available firewall configuration has lower expected residual utility than the secure configuration.

Machine	**Time 1**	**Time 2**	**Time 3**
Small	39.5	37.8	38.0
Medium	38.8	38.1	38.6
Large	36.2	36.3	36.6

Table 5.4: Time in seconds to evaluate secure firewall configuration.

Machine	**Time 1**	**Time 2**	**Time 3**
Small	34.4	33.1	33.1
Medium	34.9	34.9	35.0
Large	33.5	31.5	31.2

Table 5.5: Time in seconds to evaluate available firewall configuration.

Chapter 6

Discussion and Conclusion

The approach described in this paper addresses our motivation to automate graceful degradation of systems in response to attacks. Our implementation is a proof-of-concept that can be extended in the future. As outlined in the thesis statement in Section 1.3, we have multiple goals, including realism, effectiveness, practicality, and performance. These properties often compete with each other. For example, a more realistic model may result in decreased performance by expanding the amount of computation necessary for each evaluation; it could also result in a less usable interface by overwhelming a user with requirements for a detailed model description.

In previous chapters, we made a number of assumptions and decisions to balance these against each other. In this section, we discuss the key assumptions and decisions we made that were not discussed elsewhere in this paper. For each, we explain the limitation that created the trade-off, why we made the decision the way we did, what it would take to compensate for or avoid the trade-off, and why our trade-off is appropriate for secure graceful degradation.

6.1 Key Assumptions and Decisions

6.1.1 Temporality

Our approach does not explicitly address temporality. We assume near-instantaneous actions and reactions (i.e., attack scenarios and defensive Courses of Action (COAs)). Therefore, our model does not include considerations like the time it takes an attacker to exploit a system, crack a password, etc.; similarly, we do not model how long a defender may take to mount a defense like installing and configuring a firewall or modifying network topology. We also do not model multiple turns as in a game-theoretic analysis. Although temporal considerations can be important for understanding whether a defense can occur fast enough to counter an attacker, we argue that our approach still provides significant value without them.

Our approach utilizes architecture views to represent the system component allocations, information flows, and system functionality. These views do not have temporal attributes, and we do not have a dedicated temporal view. Additionally, our evaluation of attack scenarios and defensive COAs do not include notions of time. The addition of time as first class would create significant new complexity in both how it is represented and how it is evaluated.

There are two primary ways to represent temporality: In one way, time is represented by “turns” taken between the attacker and defender; this would be compatible with game theory. The other way, time is represented in absolute terms along a timeline; for example, a particular exploit could take 2.7 seconds, and a subsequent response might take another 3.1 seconds.

A turn-based (i.e., game theory-like) approach requires more assumptions about the attacker than our approach, which assumes a worst-case scenario attack. A temporal approach may similarly make some assumptions about the amount of time that specific actions take. One could extend the implementation of our approach to include the notion of turns, though an attacker would have to be modeled with more specificity, such as having a specific goal. Running our approach multiple times would reduce performance while requiring additional model specification.

In a timeline-based approach, it would be difficult – but possible – to include time as a constraint just like attacker capability; the search space for attack traces might then extend only as far as a predefined time limit allows for the attack before the defender has an opportunity to reassess and react. One would need to consider whether the attacker capability can be recharged with time or if it monotonically decreases. This increases the complexity of the model specification and the analysis.

We assume the worst about an attacker – they will instantaneously attempt the most damaging attack they are capable of – and we defend for that. We assume that defenders generally wish to evict attackers as quickly as possible from systems, so we treat graceful degradation as a brief, temporary state rather than as a series of turns that take place over extended periods of time.

6.1.2 Cyberphysical Destruction

Our approach treats dataflows as the entity to be protected rather than components. In some cases, that might not be the correct assumption. For example, an internet-connected light bulb loses its utility if an attack manages to brick the bulb so that it will never boot or light up again. In a sense, the physical bulb itself loses availability from the perspective of a human user. While not perfect, a workaround is to treat the production of light as a dataflow producer with an availability attribute that is reduced during an attack. A human is then also modeled as the consumer of that "dataflow" with an availability requirement represented in some sort of function (e.g., "reading a book") with an associated utility.

6.1.3 Decoys and Deception

Our approach does not model decoys (e.g., honeypots and honeynets) or deception. These tactics can slow an attacker and increase their observability so that the defender can detect and evict them. Attackers spend time and resources attacking decoys. However, we chose to keep our model simpler and assume a worst-case scenario, in which the attackers avoid the decoys altogether. An extension of our approach could include decoys and deception by modifying the attack scenario evaluation algorithm to be, for example, something like an expected value of attack scenarios rather than the worst case scenario. By including attack scenarios through decoys, the inclusion of decoys results in a number of unproductive or limited damage scenarios. However, attackers may be smart enough to spot and avoid these decoys, so a worst case scenario should be modeled, too.

6.1.4 Exfiltration of Data

Our approach considers the confidentiality of a dataflow to be compromised the moment an attacker accesses it. This simplifies the modeling and assumes the worst case scenario. Therefore, data exfiltration does not change the utility of the system once the associated dataflow is compromised. However, some may wish to model exfiltration in a way that clearly shows how an attack can go from bad to worse. One way to do this would be to include a notion of attacker functionality, parallel to the defender's functional view. Attacker functionality would result in negative utility, and the exfiltration of data would contribute to the attacker functionality. This requires making additional assumptions about the attacker (e.g., the attacker will ask for ransom for the data) that we do not require.

In addition, we need to include the components and connectors for exfiltration in our model. The ability to add and remove connections is something we currently reserve for the defender since the defender is the one presumably in control of their network architecture, but a more robust attacker model would include an ability for an attacker to change an architecture within pre-specified constraints (e.g., establishing an exfiltration connector is allowed, modifying the system under test internal architecture is disallowed).

Although this addition would increase realism, we believe that our model is sufficient without this additional complexity. It is reasonable to assume that the data is compromised at its initial access prior to exfiltration. Additionally, exfiltration could occur surreptitiously so that the defender is uncertain as to whether or not it has occurred. Therefore, a cautious, worst case scenario approach is warranted.

6.1.5 Attacker Model

Our approach allows for simultaneous consideration of multiple attackers by allowing multiple possible points of initial presence and a range of attacker capabilities in the form of a Probability Density Function (PDF). For example, using our running example, we could consider a low probability insider threat at the same time as a high probability internet-based threat. As a simplification, the attacker capability PDF is considered to be the same regardless of where the attack scenario originates. This keeps our modeling simpler than if we considered multiple Tactics, Techniques, and Procedures (TTPss). However, it may be more realistic to consider these two or more attackers as different types of threats with different TTPss and capabilities. This would require some cross-cutting changes to our implementation to associate a capability PDF with each compromised component rather than as a global value associated with a single attacker type.

We decided to use a PDF to describe attacker capabilities because we wanted to ensure that we could address both the uncertainty for any given attacker and the different capabilities of different attackers. We use a capability rather than saying an attacker has specific exploits available *a priori* because we believe that this is more realistic than one single capability value. As the Economist and reports on Pegasus spyware have shown, there is a thriving commercial industry that develops exploits on demand for attackers, and exploits have a variety of prices [29].

An attacker goal can be useful for constraining attack traces. However, we erred on the side of caution and high uncertainty. If attacker goals can be anticipated, they could significantly improve scalability and performance.

As attacks are in progress, we anticipate that the points of presence and capabilities will be updated in the model to reflect the ongoing attacks, showing high probabilities of compromise in some components and lower ones in others. The capability PDF may also reflect increased confidence in knowledge of the capability of the current attacker rather than a distribution of probabilities of capabilities among multiple possible attackers. A defender can achieve this by updating the attacker data and rerunning the evaluation as new information is received.

6.1.6 Courses of Action

For COAs, we assume that each particular tactic has a cost that is independent of the other tactics. Together, this cost becomes the cost of the COA. For simplicity, much of our modeling has assumed an equal cost for tactics of adding and removing connections. While our approach allows for defining different costs for different types of tactics (e.g., adding a connection could cost more than removing one), we do not have the ability to say, for example, that one *particular* connection will cost more than another one to add (or remove).

One could extend the model to include more fine-grained cost differences for tactics. However, this could be very complex. As an example using connection costs, one would need to separately determine and define the cost of each possible connection. For n possible components in the model, the number of connections can grow on the order of n^2. This would be very difficult to scale.

A coarser option is to have different connection types like Local Area Network (LAN) and Wide Area Network (WAN) connections where all LAN connections would have one cost and all WAN connections would have a different cost. For each component, we would need to specify the type and an instance in the case of a LAN, since there could be multiple separate LANs in the model. The architecture style would neet to be modified to constrain LAN connections to be between two components of the same LAN instance. Only WAN connections could connect two LANs together. While this is less difficult to scale than the fine-grained option, it still adds significant complexity to our modeling. It also may result in the placement of unnecessary constraints (e.g., LAN separation strategies) on the model, reducing the ability of our approach to search creative areas of the search space that could include COAs like combining or splitting LANs.

Although this may limit the richness, we believe that our approach has yielded useful results. It allows for searching novel architectures while keeping our model as simple as possible. As we demonstrated in Chapter 5, our approach is able to reason through effective responses to attacks.

6.1.7 Dataflows

We choose to protect dataflows rather than components. Our reasoning is that in most information systems, the data is what is most critical. The compromise of a server is not the primary issue – what happens to the data on it is. This extends to cyberphysical systems.

We use dataflows because they are critical to understanding the impacts to functions. An asset-centric approach may miss this. We use data consumers and producers because this is more realistic and dynamic than if we hard coded the data flows into our models. This allows us to evaluate the impacts of adapting dataflows to use backup producers. (This may also be easier to model, as it does not require hand coding each dataflow.)

Our knowledge requirements for producers and consumers are minimal. The producer does not necessarily know the use case for the data it produces. For example, weather forecasts could be used to determine whether or not to bring an umbrella (low impact) or whether or not it is safe to fly a helicopter (high impact). There can be multiple data producers. The consumer knows how it is using the data it consumes. The consumer component is agnostic as long as security requirements are met.

6.1.8 Functionality Instead of Criticality

We chose to include a functional view rather than assign specific utility values to components. We believe this provides increased richness, though it comes at the cost of requiring a view that software engineers may find unfamiliar.

The functional approach is superior to one based on assets values or criticality. We avoid the pitfalls of focusing on big budget line-items while small components can be an Achilles' heel. A functional view allows us to realistically model how impacts can cascade.

Only some functions have an associated utility. This is because some functions do not provide any real world value on their own; they must be combined with other functions to provide a value. Taking an engine off a two engine airplane does not reduce the utility of that aircraft – it eliminates it. A similar conceptual approach applies to system functionality.

6.1.9 Implementation in Datalog

We chose to implmement Defensive degradation of Resilient Architectures (DORA) in Datalog for several reasons. First, it has been used successfully in prior research for generating attack traces in situations where vulnerabilities are known in advance. We rely on Datalog to inductively build attack scenarios leveraging hypothesized vulnerabilities, to construct data flows, and to evaluate the implications of the attack scenarios on data flows and system functionality. Datalog provides evidence for its outputs in ways that can be clearer for defenders to understand than alternatives like machine learning, in which some of the "under the hood" evaluation may not be explainable.

Other alternative approaches include game theory and satisfiability (SAT) solvers. Game theory is a very different type of analysis from our approach. Game theory, by definition, requires that the players (i.e., the attacker and defender) take turns. We assume a worst case scenario of an instantaneous attack. Similarly, we expect defenders to act as quickly as possible – without giving the attacker time to adjust their approach mid-defense. If we relax these assumptions, a game theoretic approach may add value. For example, it could address how an attacker might respond to an incomplete eviction of the attacker, or it could address how a defender might observe and learn about an attacker's acpttps so a subsequent eviction is more likely to be successful.

A game theoretic approach is still incomplete without the contributions of our approach. It does not provide a framework to understand how an attacker can move through a network, how a particular attack scenario has a cascading effect on system functionality, or how a specific COA can impact an attack scenario. The core of our approach provides all three. In that sense, game theory can build on and extend our approach, but it is not a substitute.

An alternative approach is to use a SAT solver to find a "solution" to the constraints in our model (e.g., ensuring data flows maintain the necessary security attributes despite attack).

However, this requires a special type of SAT solver with the ability to gradually weaken the constraints until the next-best solution is found. Others are researching this topic [25]. A SAT solver would replace the part of our approach that searches for defensive COAs, but like the game theoretic approach, it would still require much of the same evaluation of how attack scenarios impact system functionality – though this could be performed with an alternative language to Datalog. The theoretical underpinnings of our approach would remain while the implementation specifics would change.

While Datalog works well for our use cases, it has drawbacks. It was not built for speed in the way that tools like SAT solvers are. It can consume enormous amounts of processing time and memory if it is not sufficiently constrained. It is not purpose-built for rapid searching and finding optimal solutions. The facts and many rules may be easier to for non-experts to understand than game theory or SAT solvers, but there is a performance cost. While we believe Datalog is excellent for proving our concept elicited in our thesis statement, other tools may make this approach more practical in a real-world setting.

6.1.10 Availability of Views Documentation

We believe that much of the information our approach requires will be available in mature (in terms of process) software projects. These projects are critical enough for which graceful degradation is an important property, and this criticality should also result in additional consideration for software architecture.

The views we use are adapted from the types of views commonly used in industry. The exception is the functional view, which is less common, though it can be found in some architecture frameworks like the Department of Defense Architecture Framework (DoDAF) [55]. The functional view – while less commonly used – is based on information that is and must be readily available to a system architect or administrator, regardless of whether or not the information has been documented before.

6.1.11 Abilities of SMEs

Our sensitivity analysis shows that when Subject Matter Experts (SMEs) inputs are "in the ballpark" of each other, our approach should produce similar outputs without wild oscillations. However, the follow up is whether SMEs can achieve even that. We believe this is possible.

The cost or difficulty in exploiting components can be derived from the historical black market street value of exploits for those components. For components without known exploits, comparisons can be made to those with known exploits. These comparisons are made based on properties like relative vulnerability history and attack surface.

6.1.12 Effectiveness

We argue that we can begin with a small set of basic principles and create best practice, systematically apply them, and create secure architecture patterns as a direct result. The principles include the use of attack traces as a way to measure the security of a system architecture. These basic principles are not specific to a particular use case, so we believe they are extensible to a variety of system architectures.

6.1.13 Validation

While challenges could be brought against our approach to validation, we believe our evaluation demonstrated the value of our approach. One could argue might have an unrepresentative set of small architectures or that our architectures might not be a representative sample. For these evaluations, we used architectures that demonstrated multiple security best practices.

Our idea of what it means to be more secure for our small architectures might not be as universal as we think. Our approach to convincing the reader through the use of small examples might be flawed and not scale. We believe security best practices are a reasonable source of "ground truth." This is analogous to using SMEs to evaluate if systems are more or less secure. As systems scale, our approach can provide quantitative artifacts with outputs so the artifacts can be used to explain how one architecture may be more robust to attack than another.

6.1.14 Design and Run Time Integration

We addressed the design and run time differences in the CONOPS in Section 3.2. At design time, the source of information is almost completely manual, but at run time, it may be possible (with integrations to system administration tooling) to source some information about system state automatically. Some changes in assumptions – for example, the cost to modify architectures, the expectations for attackers – will still need to be provided manually on an as-needed basis to reflect changes and ensure outputs match reality.

6.2 Future Work

There are many possibilities for extending our work. Currently, manual or "table top" analyses can use the same inputs as our approach, including the same architecture views and attacker and defender profiles. As we show in our evaluation, this information can be used to determine attack scenarios and subsequent impacts to system functionality and utility. Improving scalability is the next step with the most potential to improve the impact of our research. Once our approach can scale to larger systems with more components, improvements to usability and explainability will make our approach more practical and valuable.

6.2.1 Scaling

Perhaps the largest factor in using our approach today in a production environment is its ability to scale to provide near real time evaluations in production-sized models. This is because of our use of Datalog and our strategy for searching the state space of COAs.

Other strategies for efficiently searching for run time COAs are left to future work. Possibilities include use of genetic algorithms, better-bounded search (e.g., if attacker goals are known), identifying and pruning degenerate architectures before evaluating them (if possible). Non-optimal searches might be a good tradeoff of performance and optimality. It may be that the state space could be efficiently searched through the use of emerging computing paradigms like quantum computing.

Additionally, a compositional divide-and-conquer approach may have the potential to increase scalability. In some circumstances, sets of components may be abstracted into a single component. In contrast, components may be refined into constituent components as necessary for an analysis focused on that higher-level component. Careful abstraction – with refinement as necessary – may enable evaluations of a system one subsystem at a time so evaluation can limit the exponential nature of the analysis.

6.2.2 Usability

Although a Graphical User Interface (GUI) would help in the generation of formal models of the views and is viewed as a possible extension of our core research in this thesis, we currently require a human to write the description of the architecture views in Datalog. A GUI would make this process much easier for system architects and administrators. It would also reduce the likelihood of syntax and other manual errors.

Automation would also improve usability in the run-time use case. System administration tools such as network scanning tools can be used to automate the discovery of components and connections on a network, providing real-time updates as the system architecture changes. Security Information and Event Management (SIEM) tools are sources of information such as suspected or known points of compromise.

As security monitoring infrastructure detects attacks, future integrations could ensure the tool automatically responds to known compromises of components by updating and rerunning the analysis. SMEs can also manually refine the attacker capability and location estimates in the model as updated threat intelligence emerges. The cost to exploit components can also be manually or automatically (with future work) refined as new information arrives. Updates to the model require a rerun of the evaluations with the new inputs. Re-analysis outputs the degraded architecture to which the current architecture should reconfigure, along with hypothetical attack trace data for explanatory and evidentiary purposes. The adaptation can be carried out either automatically (with future integration) as an automated COA (Automated Course of Action (ACOA)) or manually with a human in the loop as a COA.

Some parts of our approach are difficult to automate but are reusable. For example, the development of architecture styles requires reasoning that goes beyond translating data from one format to another, but once a style is created, it can be reused across similar systems with minimal to no changes. We believe that this process can be eased through the creation of examples and templates, significantly reducing the learning curve to apply our approach.

6.2.3 Explainability

Our approach and tool produce detailed artifacts, so explanations can be produced using the evidence provided. We are not subject to the same explainability limitations of "black box" types of machine learning algorithms. Information in the form of facts is created and stored through the application of Datalog rules, so it may require cross-cutting changes to the code if the currently-tracked artifacts are insufficient. However, the information that is currently tracked can be formatted with a small effort to provide useful evidence for explaining the COA outputs.

6.2.4 Tabula Rasa Architecture Generation

One of the more difficult scenarios for a system architect could be to produce the initial architecture for a system, since the possibilities are seemingly limitless when starting from a clean slate. Our use of Datalog makes it difficult for us to produce an initial architecture from nothing, since our assumption is to begin with an initial architecture and produce alternatives to it. Other techniques like Satisfiability (SAT) solvers could produce options for starting points, and our approach could then compare those options to each other and similar architectures.

6.3 Conclusion

In our research, we set out to assist system defenders who wish to build and operate systems that can gracefully degrade when attacked. Tooling has not kept up with the threat, and evaluating how to respond to attacks is still mostly dependent on manual (i.e., human) labor. In part, other approaches to reasoning about graceful degradation do not embrace levels of uncertainty common to real world scenarios, they do not incorporate first class notions of data flows and complex system functionality, and they are not formalizable and automatable. We describe our approach to solving this problem and demonstrate that – even with such incomplete knowledge – we can meaningfully evaluate architectures for resilience to attacks and provide defenders with COAs for preserving critical system functionality.

Chapter 7

References

Bibliography

[1] S. R. Goerger, A. M. Madni, and O. J. Eslinger, "Engineered resilient systems: A DoD perspective," Procedia Computer Science, vol. 28, pp. 865–872, 2014, ISSN: 1877-0509. DOI: 10.1016/j.procs.2014.03.103.

[2] D. Garlan and M. Shaw, "An introduction to software architecture," Carnegie Mellon University, Tech. Rep. CMU-CS-94-166, 1993. [Online]. Available: https://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf.

[3] "Cybersecurity management and oversight at the jet propulsion laboratory," NASA Office of the Inspector General, Tech. Rep. IG-19-022, Jun. 2019. [Online]. Available: https://oig.nasa.gov/docs/IG-19-022.pdf (visited on 09/14/2020).

[4] "NASA systems engineering handbook," [Online]. Available: https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook.

[5] Security content automation protocol undefined CSRC. [Online]. Available: https://csrc.nist.gov/projects/security-content-automation-protocol (visited on 12/01/2020).

[6] C. P. Shelton, P. Koopman, and W. Nace, "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems," Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003. pp. 156–163, 2003. DOI: 10.1109/words.2003.1218078.

[7] G. Fairbanks and D. Garlan, Just Enough Software Architecture: A Risk-driven Approach. Marshall & Brainerd, ISBN: 9780984618101. [Online]. Available: https://books.google.com/books?id=ITsWdAAzVYMC.

[8] Palo alto networks CEO stresses modern, integrated cybersecurity. [Online]. Available: https://www.cnbc.com/2023/08/21/palo-alto-networks-ceo-

stresses-modern-integrated-cybersecurity.html (visited on 08/22/2023).

[9] J. Holmes and G. Pruitt, "Assessment of the NASA flight assurance review program," ARINC Research Corporation, Annapolis, MD, Tech. Rep., 1983. [Online]. Available: https://ntrs.nasa.gov/api/citations/19840015333/downloads/19840015333.pdf.

[10] O. Saydjari, Engineering Trustworthy Systems: Get Cybersecurity Design Right the First Time. McGraw-Hill Education, ISBN: 9781260118186. [Online]. Available: https://books.google.com/books?id=bEBiDwAAQBAJ.

[11] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: A logic-based network security analyzer," in Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, ser. SSYM'05, Berkeley, CA, USA: USENIX Association, 2005, p. 8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251406.

[12] "Fault tree handbook," U.S. Nuclear Regulatory Commission, Tech. Rep. NUREG-0492, 1981. [Online]. Available: https://www.nrc.gov/docs/ML1007/ML100780465.pdf.

[13] T. Ishimatsu, N. Leveson, J. Thomas, M. Katahira, Y. Miyamoto, and H. Nakao, "Modeling and hazard analysis using STPA," in Proceedings of the 4th IAASS Conference, Making Safety Matter, ser. ESA Special Publication, vol. 680, Huntsville, Alabama, USA: International Association for the Advancement of Space Safety (IAASS), 2010, p. 31, ISBN: 978-92-9221-244-5. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/79639/Leveson_Modeling%20and%20hazard.pdf?sequence=2&isAllowed=y (visited on 09/24/2020).

[14] W. Young and N. Leveson, "Systems thinking for safety and security," in Proceedings of the 29th Annual Computer Security Applications Conference, ser. ACSAC '13, New York, NY, USA: Association for Computing Machinery, 2013, pp. 1–8, ISBN: 9781450320153. DOI: 10.1145/2523649.2530277. [Online]. Available: https://doi-org.proxy.library.cmu.edu/10.1145/2523649.2530277.

[15] "Mission-based risk assessment process for cyber (MRAP-c) guidebook," US Air Force, Tech. Rep. Version 2.1, 2010.

[16] M. Stamatelatos, H. Dezfuli, G. Apostolakis, et al., "Probabilistic risk assessment procedures guide for NASA managers and practitioners," NASA, Tech. Rep., 2011. [Online]. Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120001369.pdf.

[17] J. G. Rivera, A. A. Danylyszyn, C. B. Weinstock, L. R. Sha, and M. J. Gagliardi, "An architectural description of the simplex architecture.," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep., 1996. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12521.

[18] G. Klein, J. Andronick, K. Elphinstone, et al., "Comprehensive formal verification of an OS microkernel," ACM Transactions on Computer Systems (TOCS), vol. 32, no. 1, p. 2, 2014, ISSN: 0734-2071. DOI: 10.1145/2560537.

[19] C. Hawblitzel, J. Howell, M. Kapritsos, et al., "IronFleet: Proving practical distributed systems correct," in Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2015. DOI: 10.1145/2815400.2815428.

[20] O. M. Sheyner, "Scenario graphs and attack graphs," Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2020-05-08, Ph.D. dissertation, 2004.

[21] S. Jha and J. M. Wing, "Survivability analysis of networked systems," Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, pp. 307–317, 2001. DOI: 10.1109/icse.2001.919104.

[22] L. Wang, M. Albanese, and S. Jajodia, "SpringerBriefs in computer science," 2014, ISSN: 2191-5768. DOI: 10.1007/978-3-319-04612-9.

[23] D. Seto, B. Krogh, L. Sha, and A. Chutinan, "Dynamic control system upgrade using the simplex architecture," IEEE Control Systems, vol. 18, no. 4, pp. 72–80, 1998, ISSN: 1066-033X. DOI: 10.1109/37.710880.

[24] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 132–141, 2009, ISSN: 2157-2305. DOI: 10.1109/seams.2009.5069082.

[25] C. Zhang, "Behavioral robustness of software system designs," 2025. DOI: 10.1184/r1/28500467.v1. [Online]. Available: https://kilthub.cmu.edu/articles/thesis/Behavioral_Robustness_of_Software_System_Designs/28500467.

[26] W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie, "Fault tree analysis, methods, and applications – a review," IEEE Transactions on Reliability, vol. R-34, no. 3, pp. 194–203, 1985, ISSN: 0018-9529. DOI: `10.1109/tr.1985.5222114`.

[27] P. L. Goddard, "Software FMEA techniques," Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Q pp. 118–123, 2000, ISSN: 0149-144X. DOI: `10.1109/rams.2000.816294`.

[28] D. J. Reifer, "Software failure modes and effects analysis," IEEE Transactions on Reliability, vol. R-28, no. 3, pp. 247–249, 1979, ISSN: 0018-9529. DOI: `10.1109/tr.1979.5220578`.

[29] "The digital arms trade; cyber-security," English, The Economist, vol. 406, no. 8829, pp. 65–66, 2013, Copyright - (Copyright 2013 The Economist Newspaper Ltd. All rights reserved.; Document feature - Charts; Last updated - 2017-11-20; CODEN - ECSTA3. [Online]. Available: `https://search-proquest-com.proxy.library.cmu.edu/docview/1321932258?accountid=9902`.

[30] D. Garlan, F. Bachmann, J. Ivers, et al., Documenting Software Architectures: Views and Beyond, 2nd. Addison-Wesley Professional, 2010, ISBN: 9780321552686.

[31] "ISO/IEC/IEEE international standard - software, systems and enterprise – architecture processes," ISO/IEC/IEEE 42020:2019(E), pp. 1–126, 2019. DOI: `10.1109/ieeestd.2019.8767004`.

[32] M. Boddy and T. Dean, "Solving time-dependent planning problems," in Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2, ser. IJCAI'89, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 979–984.

[33] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, "Guide to industrial control systems (ICS) security," DOI: `10.6028/nist.sp.800-82r2`.

[34] "Secure data transfer guidance for industrial control and SCADA systems," Pacific Northwest National Laboratory, Tech. Rep. [Online]. Available: `https://www.pnnl.gov/main/publications/external/technical_reports/PNNL-20776.pdf` (visited on 02/08/2023).

[35] K. Hardy, Enterprise Risk Management A Guide for Government Professionals, eng. Hoboken: Wiley, 2014, ISBN: 1-118-91112-1.

[36] B. I. Koerner, Inside the OPM hack, the cyberattack that shocked the US government undefined WIRED, Oct. 2016. [Online]. Available: `https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/` (visited on 03/24/2025).

[37] N. N. Taleb, The black swan : the impact of the highly improbable (Business book summary), eng, First Edition. New York: Random House, 2007, ISBN: 9781400063512.

[38] T. M. Chen and S. Abu-Nimeh, “Lessons from stuxnet,” Computer, vol. 44, no. 4, pp. 91–93, 2011, ISSN: 0018-9162. DOI: `10.1109/mc.2011.115`. [Online]. Available: `http://dx.doi.org/10.1109/MC.2011.115`.

[39] P. K. Manadhata and J. M. Wing, “An attack surface metric,” IEEE Transactions on Software Engineering, vol. 37, no. 3, pp. 371–386, 2011, ISSN: 0098-5589. DOI: `10.1109/tse.2010.60`.

[40] NVD - home. [Online]. Available: `https://nvd.nist.gov/` (visited on 12/21/2020).

[41] Pcarbonn/pyDatalog: A datalog implementation in python. [Online]. Available: `https://github.com/pcarbonn/pyDatalog` (visited on 11/28/2023).

[42] PRISM - probabilistic symbolic model checker. [Online]. Available: `https://www.prismmodelchecker.org/` (visited on 11/30/2023).

[43] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of probabilistic real-time systems,” in Proc. 23rd International Conference on Computer Aided Verification (CAV’11), ser. LNCS, vol. 6806, Springer, pp. 585–591. DOI: `10.1007/978-3-642-22110-1_47`.

[44] D. Jackson, “Alloy,” Communications of the ACM, vol. 62, no. 9, pp. 66–76, 2019, ISSN: 0001-0782. DOI: `10.1145/3338843`.

[45] R. Ross, V. Pillitteri, R. Graubart, D. Bodeau, and R. McQuaid, “Developing cyber-resilient systems :,” 2021. DOI: `10.6028/nist.sp.800-160v2r1`.

[46] Recommended cybersecurity practices for industrial control systems. [Online]. Available: `https://www.cisa.gov/sites/default/files/publications/Cybersecurity_Best_Practices_for_Industrial_Control_Systems.pdf` (visited on 09/14/2022).

[47] About security zones. [Online]. Available: `https://www.cisco.com/assets/sol/sb/isa500_emulator/help/guide/ag1463340.html` (visited on 09/20/2022).

[48] Recommended practice: Improving industrial control system cybersecurity with defense-in-depth strategies. [Online]. Available: `https://www.cisa.gov/uscert/sites/default/files/recommended_practices/NCCIC_ICS-CERT_Defense_in_Depth_2016_S508C.pdf` (visited on 09/21/2022).

[49] C. P. Shelton, "Scalable graceful degradation for distributed embedded systems," Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2020-05-08, Ph.D. dissertation, 2003.

[50] Secure architecture design undefined CISA. [Online]. Available: `https://us-cert.cisa.gov/ics/Secure-Architecture-Design` (visited on 09/13/2020).

[51] I.-C. MIHAI, Ș. PRUNĂ, and I.-D. BARBU, "Cyber kill chain analysis," International Journal of Information Security and Cybercrime, vol. 3, no. 2, pp. 37–42, 2014, ISSN: 2285-9225. DOI: `10.19107/ijisc.2014.02.04`. [Online]. Available: `https://www.lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/Gaining_the_Advantage_Cyber_Kill_Chain.pdf`.

[52] "MITRE ATT&CK: Design and philosophy," MITRE, Tech. Rep. MP180360R1, 2020. [Online]. Available: `https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf`.

[53] P. E. Kaloroumakis and M. J. Smith, "Toward a knowledge graph of cybersecurity countermeasures," MITRE, Tech. Rep., 2020. [Online]. Available: `https://d3fend.mitre.org/resources/D3FEND.pdf`.

[54] RapidEye - eoPortal directory - satellite missions. [Online]. Available: `https://directory.eoportal.org/web/eoportal/satellite-missions/r/rapideye` (visited on 02/16/2022).

[55] DODAF - DOD architecture framework version 2.02 - DOD deputy chief information officer, Mar. 2011. [Online]. Available: `https://dodcio.defense.gov/Library/DoD-Architecture-Framework/` (visited on 09/02/2020).

Appendix A

Acronyms

Acronyms

ACOA Automated Course of Action

C&C Component-and-Connector

CAN Controller Area Network

CCSDS Consultative Committee for Space Data Systems

COA Course of Action

DORA Defensive degradation of Resilient Architectures

ETA Event Tree Analysis

FMEA Failure Mode and Effects Analysis

FMECA Failure Mode, Effects, and Criticality Analysis

FTA Fault Tree Analysis

GUI Graphical User Interface

HMI Human Machine Interface

ICS Industrial Control System

IG Inspector General

JPL Jet Propulsion Laboratory

LAN Local Area Network

MRAP-C Mission-based Risk Assessment Process for Cyber

MSL Mars Science Laboratory

NASA National Aeronautics and Space Administration

NIST National Institute of Standards and Technology

OPC Open Platform Communications

PDF Probability Density Function

PDR Preliminary Design Review

PRA Probabilistic Risk Assessment

RQ Research Question

RTU Remote Terminal Unit

SAT Satisfiability

SCADA Supervisory Control And Data Acquisition

SIEM Security Information and Event Management

SME Subject Matter Expert

SMT Satisfiability Modulo Theories

STPA Systems Theoretic Process Analysis

STPA-Sec Systems Theoretic Process Analysis-Security

TTPs Tactics, Techniques, and Procedures

VPN Virtual Private Network

WAN Wide Area Network

Appendix B

ICS Exemplar System Description in Datalog

```
#Components, Types, and Vulnerabilities
+ isType('opc','opcT')
+ isVulnerable('opcT','opcExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('opc',['admin'])
+ usesCredential('opc','admin')

+ isType('hmi','hmiT')
+ isVulnerable('hmiT','hmiExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('hmi',[])

+ isType('scada','scadaT')
+ isVulnerable('scadaT','scadaExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('scada',[])

+ isType('engineerWorkstation','engineerWorkstationT')
+ isVulnerable('engineerWorkstationT','engineerWorkstationExploit'
    ,1.0,0.0,0.0,0.0)
+ hasCredentials('engineerWorkstation',[])

+ isType('historian','historianT')
+ isVulnerable('historianT','historianExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('historian',[])

+ isType('ntp','ntpT')
+ isVulnerable('ntpT','ntpExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('ntp',[])

+ isType('switchA','switch')
+ isVulnerable('switch','switchExploit',0.0,1.0,1.0,1.0)
+ hasCredentials('switchA',[])

+ isType('switchB','switch')
+ hasCredentials('switchB',[])

+ isType('dmzFirewall','firewallT')
+ isVulnerable('dmzFirewallT','dmzFirewallExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('dmzFirewall',[])

+ isType('printer','printerT')
+ isVulnerable('printerT','printerExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('printer',[])

+ isType('secondaryHistorian','historianT')
+ hasCredentials('secondaryHistorian',[])

+ isType('vpn','vpnT')
+ isVulnerable('vpnT','vpnExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('vpn',[])

+ isType('relay1','relayT')
+ isVulnerable('relayT','relayExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('relay1',[])

+ isType('relay2','relayT')
+ hasCredentials('relay2',[])

+ isType('rtus','rtuT')
+ isVulnerable('rtuT','rtuExploit',1.0,0.0,0.0,0.0)
+ hasCredentials('rtus',[])

# Attacker Profile
probCapability[0.0] = 0.2
probCapability[1.0] = 0.2
probCapability[2.0] = 0.2
probCapability[3.0] = 0.2
probCapability[4.0] = 0.2
+ compromised('vpn',1.0,True,True,True)
+ compromised('printer',0.5,True,True,True)

# Data Flows
+ producesData('hmi','setPointsRestData')
+ consumesData('hmiF',['hmi'],'statusRestData',0.0,0.75,0.25)


+ consumesData('monitoringF',['printer'],'statusRestData',0.0,0.75,0.25)
+ consumesData('monitoringF',['printer'],'setPointsRestData',0.0,0.75,0.25)

+ consumesData('relay1F',['relay1'],'actionsModbusData',0.0,0.75,0.25)

+ consumesData('relay2F',['relay2'],'actionsModbusData',0.0,0.75,0.25)

+ producesData('rtus','statusModbusData')

+ consumesData('scadaF',['scada'],'actionsRestData',0.0,0.75,0.25)
+ consumesData('scadaF',['scada'],'statusRestData',0.0,0.75,0.25)
+ consumesData('scadaF',['scada'],'setPointsRestData',0.0,0.75,0.25)
+ producesData('scada','actionsRestData')

+ consumesData('workstationF',['engineerWorkstation'],'actionsRestData'
    ,0.0,0.75,0.25)
+ consumesData('workstationF',['engineerWorkstation'],'statusRestData'
    ,0.0,0.75,0.25)
+ consumesData('workstationF',['engineerWorkstation'],'setPointsRestData'
    ,0.0,0.75,0.25)

+ consumesData('historianF',['historian'],'actionsRestData',0.0,0.75,0.25)
+ consumesData('historianF',['historian'],'statusRestData',0.0,0.75,0.25)
+ consumesData('historianF',['historian'],'setPointsRestData'
    ,0.0,0.75,0.25)
+ consumesData('historianF',['historian'],'timeData',0.0,0.75,0.25)

+ producesData('ntp','timeData')

+ consumesData('opcF',['opc'],'actionsRestData',0.0,0.75,0.25)
+ consumesData('opcF',['opc'],'statusModbusData',0.0,0.75,0.25)
+ producesData('opc','actionsModbusData')
+ producesData('opc','statusRestData')

#Functions

+ utility('transmissionMgmt',50.0)
+ utility('transmission',100.0)

+ fNodeOr('transmissionMgmt',['transmission'])
+ fNodeAnd('transmissionMgmt',['transmission'])
+ fNodeAnd('transmission',['opcF','hmiF','scadaF','relaysF','historianF','
    workstationF'])
+ fNodeAnd('relaysF',['relay1F','relay2F'])
```

Listing B.1: ICS Exemplar Sample in Datalog

Appendix C

Attack Trace Generation in Datalog

```
#Base case special case for no capability
attackPaths(SourceService,SourceService,P,[],0) <= compromised(
    SourceService) & (P==[])
#Base case
attackPaths(SourceService,TargetService,P,[VulnType],TotalC) <= compromised
    (SourceService) & cToWithPrivileges(SourceService,TargetService,VulnType
    ,TotalC) & (P==[]) & (TotalC <= MaxR)#Inductive case, new exploit
attackPaths(SourceService,TargetService,P,E,TotalC) <= attackPaths(
    SourceService,IntermediateService1,P2,E2,TotalC2) & cToWithPrivileges(
    IntermediateService1,TargetService,VulnType,C) & (SourceService!=
    TargetService) & (SourceService._not_in(P2)) & (TargetService._not_in(P2
    )) & (P==P2+[IntermediateService1]) & (VulnType._not_in(E2)) & (E==E2+[
    VulnType]) & (TotalC==TotalC2+C) & (TotalC2+C <= MaxR)
#Inductive case, previously-used exploit
attackPaths(SourceService,TargetService,P,E,TotalC) <= attackPaths(
    SourceService,IntermediateService1,P2,E2,TotalC2) & cToWithPrivileges(
    IntermediateService1,TargetService,VulnType,C) & (SourceService!=
    TargetService) & (SourceService._not_in(P2)) & (TargetService._not_in(P2
    )) & (P==P2+[IntermediateService1]) & (VulnType._in(E2)) & (E==E2+[
    VulnType]) & (TotalC==TotalC2) & (TotalC2+C <= MaxR)
```

Listing C.1: Attack Trace Generation

The first base case applies to the case in which the starting point in the trace is compromised, but no others are compromised. The second base case applies to the case in which the starting point in the trace is compromised, and a vulnerability in a neighboring component allows for extension of the attack trace from the starting point to the neighbor of the starting point. There are two inductive cases. Each applies to a case in which an existing trace can be extended to an additional node. In one case, an existing exploit is reused to extend the attack trace at no extra cost to the attacker. In the other case, the attacker must use a portion of her remaining capability budget to generate a new exploit and extend the trace.

Appendix D

Algorithm for Estimated Residual Utility

```
function EstResidualUtility($a, Cap$)
    $bestCaseUtil \leftarrow$ Util($a$)
    $u \leftarrow 0$
    $a \leftarrow$ AssumeAllVulnerabilities($a$)
    for $attackerCapability = 0..max(Cap)$ do
        $T \leftarrow$ AllAttackTraces($A, attackerCapability$)
        $worstCaseUtil \leftarrow bestCaseUtil$
        for all $t \subset T$ do
            $a \leftarrow$ ApplyAttackTrace($a, t$)
            $attackScenarioUtil \leftarrow util(a)$
            if $attackScenarioUtil < worstCaseUtil$ then
                $worstCaseUtil \leftarrow attackScenarioUtil$
            end if
        end for
        $u \leftarrow u + Cap(attackerCapability) * worstCaseUtil$
    end for
    return $u$
end function
```

Algorithm 1: Residual utility estimation