Speculative Software Modification

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Benjamin D. Rodes

May 2015

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Benjamin D. Rodes

This dissertation has been read and approved by the Examining Committee:

John C. Knight, Advisor

Jack Davidson, Chair

Dave Evans

Alfred Weaver

John Lach, Minor Representative

Accepted for the School of Engineering and Applied Science:

James Aylor, Dean, School of Engineering and Applied Science

May 2015

Abstract

Speculative Software Modification (SSM) is an engineering approach for modifying software for which either minimal or no software development information and/or artifacts are available. Software of this form is commonly referred to as *Software Of Uncertain* (*or Unknown*) *Pedigree* (*or Provenance*) (SOUP). SOUP raises many doubts about the existence and adequacy of desired dependability properties (e.g., security or safety) motivating some users to apply software modifications to improve or enhance the software with respect to these properties. Without necessary development artifacts, however, modifications are made in a state of uncertainty and risk. Lack of artifacts and associated uncertainties motivating users to modify software also present uncertainties about how to effectively apply a modification: i.e., a modification might not be effective, break program semantics, or not meet other user-defined constraints.

SSM is an assurance-based engineering model instantiated by engineers to alter SOUP and address modification risks and uncertainties. The model consists of two primary components: (1) a process architecture, and (2) an *assurance case*. The process architecture provides general guidelines and activities for generating SOUP modifications. The SSM process architecture is described as an iterative process of selecting and validating *hypotheses* about how to modify a specimen of SOUP. The assurance case is used as an acceptability model to justify that any modification produced by the SSM process will be acceptable to the system stakeholders. The assurance case is a rigorous and comprehensive argument about the acceptability of SOUP modifications. Engineers instantiate both the process architecture and assurance case for their particular operating environment. Once instantiated, the process can be reused to modify any number of programs.

This dissertation presents the rationale, components, guiding principles and activities of the SSM model. Modifying software to enhance software security is currently an area of active research and presents many unique challenges. In this dissertation, I focus the application of SSM to enhance software security for illustration. Security is a composite dependability property, typically described in terms of integrity, confidentiality and availability.

SSM is evaluated through a series of case studies examining the feasibility and practicality of the SSM concept. In particular, feasibility and practicality is examined first by a case study exploring the utility and form of the SSM process architecture. A subsequent case study assesses the feasibility and practicality of applying assurance cases in SSM. A final case study examines how engineers can determine the applicability of SSM and apply SSM concepts from "first principles". These case studies are based on examination of two specimen security-enhancing modification technologies.

Dedication

To my advisor, Dr. John Knight,
for your patience and support, and the opportunities you have provided me;

To my good friends, Neil, Cris, Billy and Malcolm,
my brothers, Mike and Murray,
my sisters-in-law, Wendy and Emily,
my nieces, Morgan and Evelyn, and my nephew, Travis,
and my adopted cats, Midnight, Darla and Abel,
for your unfailing support, without which this would not be possible.

Acknowledgments

This research is supported by National Science Foundation (NSF) grant CNS-0811689, the Army Research Office (ARO) grant W911-10-0131, the Air Force Research Laboratory (AFRL) contract FA8650-10-C-7025, and DoD AFOSR MURI grant FA9550-07-1-0532. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, AFRL, ARO, DoD, or the U.S. Government.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> *"If he says profoundly, 'I never speculate', he is an ignorant speculator, and probably an unsuccessful one."* — Carret, *The Art of Speculation* [1]

This dissertation introduces Speculative Software Modification (SSM), an engineering approach for modifying software to establish one or more desired properties where either minimal or no software development artifacts are available. Development artifacts include the software source code and virtually any kind of design, development and testing documentation. Software of this form is commonly referred to as *Software Of Uncertain* (*or Unknown*) *Pedigree* (*or Provenance*) (SOUP).

Lack of software development information about SOUP raises many doubts about the existence and adequacy of desired dependability properties (e.g., security or safety), motivating some users to apply software modifications to improve or enhance the software with respect to these properties. Without necessary development artifacts, however, modifications are made in a state of uncertainty and risk. Lack of information necessary to apply an effective modification can result in modifications that:

- partially or completely fail to establish the desired property (e.g., failure to establish the desired level of safety or security),
- deviate substantially from desired or expected program behavior, or

- fail to meet other user-defined constraints and requirements (e.g., run-time or memory consumption efficiency requirements).

Ideally, users of SOUP would prefer a predefined process of SOUP modification that can be applied to a large corpus of SOUP repeatedly. If the risks of using SOUP are sufficiently high, users might be able to justify engineering such a process themselves. In practice, the uncertainties involved with SOUP modification necessitate some flexibility in how modifications are applied: some approaches are more or less applicable depending on the characteristics of the software being modified. Consequently, a predefined modification process must support generating hypotheses about how to modify the SOUP adequately, and then assess and refine these hypotheses as deemed necessary; i.e., engineers must sometimes *speculate* about how to best apply software modifications.

When engineers find themselves in a situation where modifications require speculation, they are often unaware that they are speculating and therefore address speculation in an ad hoc manner. Some engineers might be adamantly unwilling to accept and/or admit that they take part in speculation as a point of pride or reputation. They might believe speculation is nothing more than guesswork based on minimal and generally unreliable evidence. Further, they might hold that being associated with speculation would give the impression of participating in pseudoscience and generally being misinformed and unprofessional. Consequently, discussion about how to handle speculation properly is easily ignored and marginalized. The premise of this dissertation is the following:

> *Speculation is often inevitable when modifying SOUP and a structured engineering approach is necessary to provide guidance, both in realizing that speculation is occurring and how to engineer speculative software modifications.*

Speculative Software Modification explicitly addresses speculation when modifying SOUP to establish desired dependability properties. Resulting modifications are made with assurance sufficient to meet the needs of the system's stakeholders. To provide focus, this dissertation targets applying SSM to establish/enhance software *security* properties specifically. This dissertation presents the guiding principles and general mechanics of SSM.

1.1 Disambiguating the Concept of Speculation

The term *speculation* is overloaded, and is associated with different concepts. In the most general and widely applicable concept, speculation is defined as follows:

Definition 1.1. ***Speculation***: *Ideas or guesses about something that is not known* [2]. *The forming of a theory or conjecture without firm evidence* [3].

Another common use of the term occurs with respect to business and finance:

Definition 1.2. ***Financial Speculation***: *Assumption of unusual business risk in hopes of obtaining a commensurate gain* [2]. *Investment in stocks, property, etc. in the hope of gain but with the risk of loss* [3].

Within the scope of computer science, speculation is often associated with the concept of speculative execution of instructions performed by CPUs:

Definition 1.3. ***Hardware-based Speculation***: *A processor optimization technique in which the processor attempts to execute instructions that are dependent on the outcome of a branch before it is known what branch should be taken. A branch is selected and executed as if the selection is correct. If the branch prediction is later invalidated, the speculatively executed instructions are abandoned, and the correct branch is then executed* [4].

In the context of this dissertation, the concept of speculation refers to speculative decisions in modifying software to establish dependability properties (security, for example), and is defined as a combination of the above definitions:

Definition 1.4. ***Speculative Software Modification (SSM)***: *An engineering model for modifying software to effect one or more dependability properties (e.g., security, safety, reliability) but with a risk that the modified software will later be found to be unacceptable. Modifications might be unacceptable because they are based on potentially unreliable yet necessary information to produce*

the modification. SSM selects an approach for analyzing and modifying the software speculated to be correct, and assesses and refines the selection iteratively.

1.2 Security and SOUP

The general principles of SSM are likely applicable to various fields of study, and for enhancing software with respect to various dependability properties; however, modifying software to enhance software security is currently an area of active research and presents many unique challenges. As such, this dissertation focuses on the engineering principles of SSM for which the intent of software modification is to enhance software security specifically.

A software system is said to be secure if key *assets* of value are defended adequately from malicious attacks. Typically, security attacks seek to corrupt the confidentiality, integrity, and availability (CIA) of valued assets. No single and universally accepted definition of security exists. Instead, the definition of adequate security is context-specific, based on the perceived value of security-critical assets, the properties of CIA that are applicable for each asset, and, fundamentally, the needs and opinions of the software stakeholders [5].

Comprehensive information-system security is a desirable although elusive goal. Despite extensive research and the deployment of new technology, successful attacks against high-value targets continue to take place with alarming frequency. For example, recent successful software attacks on major US retailers Target, Neiman Marcus, and Michaels led to compromised customer credit card information [6, 7, 8]. Generally, software security attacks can lead to loss of:

- service,
- private/sensitive assets,
- customers/revenue, and
- reputation.

The ripple effect from a successful security attack may even have broader impacts, affecting individuals and organizations that were not the direct targets of attack. For example, a successful attack could lead to lower stock prices, affecting shareholders and potentially the stock value of similar companies. An attack could also endanger the environment or general public safety, such as an attack on a nuclear power facility.

1.2.1 Traditional SOUP Characteristics

A major security concern in modern software systems is the use of software for which minimal information is available about the development of the software [9]. At the heart of this issue is uncertainty about where the software came from and what standards were used in its development; hence, this software is often referred to as Software Of Unknown (or Uncertain) Provenance (or Pedigree), or SOUP [10]. Without supporting information and development artifacts, users of SOUP might question the quality of the software and, consequently, ask whether the software is appropriate to use.

SOUP is increasingly prevalent because of offshore software development and the widespread use of third party software components that are developed all over the world. These practices can lower software costs and potentially improve software quality by taking advantage of a vast talent pool; however, the scale and complexity of globalized software development typically makes tracking and documenting the development process difficult.

SOUP comes in a wide variety of forms, from individual code fragments, functions and libraries to entire applications and services. SOUP often takes the form of third-party libraries, open source

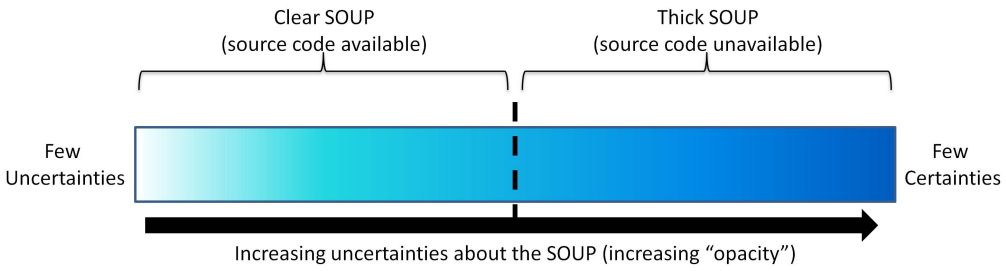


Figure 1.1: The SOUP spectrum

projects, legacy applications and commercial-off-the-shelf (COTS) software. The availability of development information for SOUP is variable, resulting in a spectrum of SOUP "*flavors*." Bishop et al. divide the spectrum into two primary flavors [11]:

Thick SOUP: Limited or no access to software development artifacts, such as source code and design documentation. Information about the development methodologies used to produce the SOUP might also be unknown (e.g., COTS software).

Clear SOUP: Source code is available but all other information and documentation might be limited or missing entirely (e.g., open source software).

Because source code facilitates easier understanding and alteration of the SOUP, the dividing line between clear and thick SOUP is the availability of source code (see Figure 1.1). At one extreme, absolutely no development information or artifacts are available about the software other than the binary form of the software (extremely thick or opaque SOUP), and, at the other extreme, most artifacts are available (extremely clear or near transparent SOUP).

1.2.2 A New SOUP Concept

The historic distinction between SOUP and any other kind of software arises from the intuition that if users have significant doubts about the quality of the software, they might rely on knowledge of where the software came from to address their concerns. That is, the provenance or pedigree of the software might provide users with sufficient confidence about the software's quality.

For example, a user might question if the software is adequately secure—was security considered during development of the software and to what extent? The user might consider the software adequately secure despite the lack of information about the software if he or she knows it came from a trusted source. For instance, the user might consider software from Microsoft acceptably secure even without adequate development information. The assumption is that a trusted source would use appropriate engineering practices to produce high-quality (in this case high-security) software. If, however, the source is unknown, then the software is referred to as SOUP.

This notion of SOUP suggests that unknown provenance is the defining characteristic, yet the term SOUP has been applied to software for which the developer is known but necessary development artifacts are still lacking or missing altogether [11]. Hence, the term SOUP can lead to some confusion and debate because of the use of "pedigree" and "provenance" in the acronym. For example, a user might know their software came from Microsoft but does not trust Microsoft. In this case, it is not clear if the software would be considered SOUP. To address this confusion, the remainder of this dissertation uses the following altered SOUP concept:

Definition 1.5. ***Software Of Unacceptable Properties (SOUP)***: *Software for which users desire one or more necessary properties (either functional or non-functional properties), but either (1) cannot determine reliably whether the properties exist and therefore the users consider the software unacceptable, or (2) the software has definitively been shown to lack the necessary properties and is unacceptable by definition.*

The intuition in this altered concept of SOUP is that the fundamental problem facing users of SOUP is not provenance or pedigree; rather, it is that users lack necessary information to determine if the software is acceptable for their use. If information allowing users to determine the acceptability of the software is not available, the software is likely to be considered *unacceptable* by a skeptical user. Software Of Unacceptable Properties includes the traditional concept of SOUP, but also includes any software for which missing information from the development process has led users to

find the software unacceptable, regardless of whether the pedigree or provenance of the software is known.

While all software systems have residual doubts and uncertainties, software is considered SOUP in the context of this dissertation only when the unknowns or uncertainties make the use of the software unacceptable to the system stakeholders. This concept also includes the possibility that the software has been definitively shown or proven to be inadequate.

1.2.3 Targeted SOUP Users

Because SOUP is practically ubiquitous and can take various forms, users of SOUP can can also come in a variety of forms, such as:

- Individuals using SOUP for private purposes,
- Developers using SOUP as a component of a product to be sold or used internally within an organization, or
- Organization, companies or government agencies in which SOUP is a component within a large *software system* consisting of numerous software applications and services.

All users of SOUP are confronted by the same concerns about the SOUP's quality; however, the target scenario for this dissertation is primarily for security-critical organizations, and potentially also for developers of security-critical software. These users encounter large quantities of SOUP, and the impacts of SOUP with poor security can be substantial. As such, these users can justify spending engineering resources to address security concerns themselves. The work presented in this dissertation is motivated by the need for and the challenges associated with using SOUP within a security-critical organization.

1.3 Making SOUP Acceptable

The potential impact of using Software Of Unacceptable Properties often motivates users to take proactive measures to establish adequate assurance that critical dependability properties are present in SOUP *prior* to deployment. This is especially the case when using SOUP in security-critical settings (as is the focus of this dissertation). In security-critical software, critical assets of the software (e.g., essential services or sensitive personal/financial information) must be defended adequately against attacks on confidentiality, integrity and availability. If the software cannot be shown to defend these assets adequately, or has been shown not to defend them, the software cannot be used in its original form.

1.3.1 Build, Buy, Examine or Modify?

One way to avoid the uncertainties associated with using SOUP is to develop the necessary software in-house from scratch. In practice, this is usually impractical because of the required engineering effort and costs. Further, the necessary algorithms to recreate the software might not be publicly available nor easily reverse engineered from the original software.

A second option is to seek acceptable alternative software, i.e., non-SOUP alternatives that have the desired properties, but this approach is also problematic. A non-SOUP alternative might not exist, might lack the necessary functionality, or might raise its own set of uncertainties about its security properties. Users might be able to find alternatives in some scenarios; however, SOUP is extremely common and might be unavoidable.

A third option is to conduct an appropriate set of analyses (testing, static analysis, etc.) on the subject program to determine if the desired property is present. If analysis can show that the software is acceptable, then the software can be used in its original form. Unfortunately, analysis might fail to show that the software is acceptable for several reasons:

1. suitable analysis techniques and tools might not exist,

2. suitable analysis techniques might not provide conclusive results,

3. the resources required to conduct the analyses might not be available, or

4. the time required to conduct the analyses might delay use of the SOUP unreasonably.

Further, even if an analysis did exist, the software might be found to be devoid of the desired property.

If users have no other recourse than to use SOUP, rather than abandoning a software solution altogether, they could attempt to achieve suitable security in the software either by:

1. altering the operating environment in which the software executes (e.g., the hardware or the operating system), or

2. directly modifying the software itself.

Software modification has the benefit of being localized to an individual program, potentially decreasing the complexity of a modification and minimizing the impact of an incorrect modification. In addition, diverse modifications are likely to be more easily deployed than using diverse hardware or operating systems. Although the techniques presented in this dissertation might generalize to modifications of the operating environment, the scope of this dissertation is the direct modification of the SOUP itself.

1.3.2 Modification Challenge 1: Sources of Uncertainty

The unknown and uncertain software characteristics motivating the need for SOUP modification also present significant challenges to modification development. There are two general categories of uncertainties and unknowns affecting the successful application of security modification techniques:

1. **Uncertain SOUP**: Lack of prior knowledge about which programs will need to be modified.

2. **Analysis Limitations**: Limitations of chosen SOUP analysis methods that lead to potentially unreliable data upon which a modification is based.

Uncertain SOUP

Prior knowledge about SOUP characteristics might be unavailable and uncertain as result of the need within an organization to apply modifications across a range of software, which might change daily. For example, an organization using hundreds of applications cannot anticipate the form of updates and upgrades that will be needed in the future. Additionally, as an organization evolves, their software needs might also evolve, requiring the use of completely new software.

Attempting to modify software in settings such as these by engineering a modification technique on an individual program basis might be impractical due to the number of applications and updates. Instead, a few modification techniques must be chosen and applied throughout the organization; however, without knowing the precise details of the software to modify, it is difficult to anticipate how effective chosen modifications will be beforehand. In essence, the issue is a potential for SOUP to have characteristics that might violate necessary assumptions for a given set of modifications to be successful. Example SOUP characteristics affecting modification include:

- Size and complexity
- Purpose
- Compilers idioms
- Testability (i.e., the extent to which the given SOUP can be tested)
- Analyzability (i.e., the ability for chosen analyses to recover data about the SOUP)
- Modifiability (i.e., the extent to which a given modification approach will be applied to the SOUP)

Analysis Limitations

Limitations in analyses occur as a result of either computational limits in what can be recovered from SOUP (i.e., analyses run up against undecidability limitations), or as a result of the practicality and

feasibility of performing a set of analyses (e.g., an analysis might require too many computational resources to produce reliable data, and therefore cannot be reasonably applied).

In essence, to generate a "perfect" modification might require perfect recovery of certain kinds of data about the software, which is often impractical or impossible. For example, without access to the source code, altering the software can be extremely difficult because high-level abstractions, such as the majority of control flow information and data structure abstractions, are lost/obscured irrevocably after compilation [12]. Even disassembly of a binary program to identify the instructions is potentially unsound [13]. Other example data that might not be reliably recovered include:

- Program architecture and organization
- Location, size, type, and purpose of data structures
- Location, size, and purpose of functions and instructions
- Control and data flow information

1.3.3 Modification Challenge 2: Risk Redistribution

Fundamentally, SOUP modifications that are designed to establish useful dependability properties and thereby reduce the risk of using the software often have their own significant risks. In essence, users of these modifications must accept risk to counter risk. The claim and purpose of SSM is that this apparent paradox can be resolved by using software modifications to redistribute risk such that the redistribution is considered acceptable whereas use of the original, unprocessed software would be unacceptable.

There are three primary software qualities of acceptability that are at risk and require balancing when modifying SOUP:

Efficacy Software modification is performed to establish one or more desired properties (e.g., security); therefore, a modification should establish these properties to the desired (acceptable) degree.

Correctness Modifications should preserve the *intended* (not necessarily original) program semantics (some modifications are made with the explicit intent of modifying program semantics; hence, these semantics alterations are "*intended*"). This implies modifications should be made (1) precisely, i.e., all components that need to be modified are modified and no others, and (2) accurately, i.e., what is modified is modified in an appropriate manner.

Efficiency The modified software should be developed and should operate using acceptable levels of resources (i.e., time, memory, hardware, etc.).

Other qualities of importance may exist but are dependent on the type of modification used and any additional constraints system stakeholders may place on the modification (e.g., modification complexity or compliance with laws and company regulations).

Stakeholders wishing to modify SOUP must assess risks to these dimensions of quality and determine an *acceptability model* that documents explicitly how the modification will balance these risks and that the balance of the risks is acceptable. Acceptability is a subjective concept and will therefore need to be tailored to the application, the modification, and the stakeholders' requirements.

1.3.4 Unique Challenges

Many of the presented motivations for and challenges of SOUP modification overlap with concepts found in traditional software maintenance activities [14]. Nevertheless, engineering SOUP modifications to establish dependability properties presents unique engineering challenges. The distinction arises for the following reasons:

- A SOUP modification is designed to establish properties throughout the SOUP that will enhance the dependability of the software. Usually, maintenance is concerned either with a local issue such as fixing a bug or enhancing a feature, or with a software-wide issue such as rehosting or accommodating an operating system upgrade.

- Maintenance is usually undertaken by a system's original developers whereas a SOUP modification is carried out by developers not connected with the SOUP's original development. Clearly, the original developers will usually have far more information than the SOUP users.

- Typically, maintenance is performed by developers as an engineering effort for one program. Conversely, developers modifying SOUP might need to alter numerous programs, both known and unknown a priori. Modifications by their very nature must be tailored to individual programs, but the purpose of SOUP modification is to establish a common property or properties for perhaps several programs leading to an *acceptable software system*.

Performing individualized maintenance activities on all SOUP encountered is an option, but the number of programs to modify might make traditional maintenance activities infeasible or impractical. Instead, stakeholders desire modification methods that are predefined and can be applied to software in an automated or semi-automated fashion.

1.4 The Speculative Software Modification Approach

Speculative Software Modification (SSM) is an engineering approach for generating SOUP modifications for which the challenges and uncertainties of SOUP modification are addressed through the cyclic generation, evaluation, and refinement of *hypotheses*. A hypothesis is a supposition about how the SOUP should be modified and is based upon information that might be unreliable, uncertain, fragmented or absent. Assessments are performed to determine the quality of hypotheses. The quality of hypotheses is assessed by an examination of the characteristics of the SOUP, the results generated by analyses, and the distribution of risks (see Sections 1.3.2 and 1.3.3). Hypotheses determined to be of sufficient quality, as defined by the system stakeholders, are considered validated and used to modify the subject program. Hypotheses can be invalidated as more data becomes available through assessment activities. An invalidated hypothesis triggers refinement of the modification.

The SSM process can be conceptualized as asking the following questions when attempting to modify a given piece of SOUP:

- What modification approach is preferred?
- What evidence would be convincing that the modification approach is acceptable?
- What alternative approaches should be considered if sufficient evidence is not available?

The SSM process is instantiated by engineers for a particular operating context as defined by system stakeholders. Once instantiated, the process can be applied to any SOUP for which modification is desired.

The term *speculation* primarily refers to the iterative process of generating, validating, and refining hypotheses about SOUP modifications; however, a validated hypothesis *does not* necessarily imply it is proven accurate, reliable, sound, precise, etc. nor proven to possess desired dependability properties. Residual doubts about the final modification might be unavoidable. In this sense, the purpose of validation is to reduce doubts *As Low As Reasonably Practicable* (ALARP) [15], not necessarily to eliminate them. As such, the term speculation also refers to the resulting validated modification, i.e., the resulting modification is speculated to be acceptable based on all available evidence.

While speculation implies a lack of definitive evidence to prove the acceptability of SSM modifications, stakeholders can still define specific conditions for which residual doubts are considered acceptably low. To provide the stakeholders with a basis for judgment about the utility and acceptability of an SSM-derived modification, SSM includes the creation of an *assurance case* [16] (described in Chapter 2). The SSM assurance case governs the manner in which acceptable modifications are produced, and provides a rationale for the acceptability of those modifications. In this respect, the SSM assurance case acts as a model of acceptability and is the core concept and driver of SSM. The construction and form of the assurance case dictates how the general SSM process is instantiated, and provides the basis for how to determine both if SSM is appropriate and why any modification produced by SSM can be considered acceptable.

The key contribution of SSM is that, to our knowledge, SSM is the first engineering approach for generating and reasoning about (i.e., analyzing and evaluating) SOUP modifications from the perspective of system stakeholders and not the original developers when essential information is unavailable. The principle benefits of SSM are:

- Modifications are designed to customize risks to allow for stakeholder-defined acceptability.
- The assurance case facilitates structured reasoning about the acceptability of produced modifications.
- Stakeholders can choose which software properties are desired, and devise custom methods for choosing, assessing and refining hypotheses in order to achieve the desired properties.
- SSM can be used to enhance and unify existing software analysis and modification techniques, as well as new sophisticated and heuristic approaches. Stakeholders can select how these methods should be applied and how they should be configured to achieve acceptable modifications.

1.5 Dissertation Thesis and Goals

In this dissertation, SSM is applied and evaluated as a method for generating security-enhancing software modifications. The main thesis of this work is that SSM is a useful approach for SOUP modifications, or more specifically:

> **Thesis:** *Speculative Software Modification is a feasible approach to produce practical SOUP modifications for establishing desired dependability properties in software.*

Feasibility refers to the realistic/reasonable application of the SSM approach and *practicality* refers to the ability of the approach to provide real benefits for engineers of software modifications. To test this hypothesis, I conducted three case studies to answer the following three questions:

1. What are the general motivation, form, rationale, and potential utility of SSM?

2. How would engineers determine if SSM is appropriate solution for a given modification technique, i.e., how would engineers find areas of weakness to enhance with SSM?

3. Once engineers determine modification weaknesses that might benefit from SSM, how are these results integrated into the SSM model?

Over the course of each case study, the concepts and guiding principles of SSM are discovered and refined. Subsequent studies exercise/illustrate and further refine discovered principles.

Case Study 1

To answer the first question, I performed an exploratory case study of SSM to expose and demonstrate the major process components and utility of SSM. The target security modification was developed and conceived for this project and is referred to as Stack Layout Transformation (SLX). The aim of this study was to investigate the feasibility and practicality of SSM by investigating the process mechanics of the SSM approach realized by SLX. This case study revealed the need for explicit models of acceptability in SSM, further investigated in the second study.

Case Study 2

Answering the second and third questions first requires an answer to the following question:

How can engineers rationalize and understand the acceptability of modifications an instantiation of SSM produces?

I performed a second case study to answer this question by examining SLX with respect to its model of acceptability. This case study explores applying assurance cases to SSM as an acceptability model for SSM. The results of this case study expose the benefits and limitations of traditional assurance case techniques when applied to SSM. I further proposed and illustrated extensions to traditional assurance case methods to address the discovered limitations, referred to as *selection argumentation*. By applying assurance cases to SSM, an explicit justification for SSM modifications can be documented and examined, thus supporting the practical application of SSM. The construction of

an SSM assurance case can therefore serve as a means for reasoning about software modifications to determine if SSM is an appropriate framework to apply, further examined in the third case study.

Case Study 3

An important result from the second case study is that an SSM assurance case not only serves to expose areas of weakness with security modifications, but also serves as a specification for an SSM process. The application of selection argumentation specifies the general SSM process mechanics within the argument itself. Hence, the development of an SSM assurance case can be used to answer the second and third research questions.

For this study, a security modification technique developed independently from this dissertation was the target of evaluation. This modification technique, referred to as S^3, is a protection mechanism for OS command injections for binary programs. In this study, I used the development of an assurance case for S^3 as the primary engineering driver for SSM. In developing the assurance case, engineers not only expose weaknesses they must address, but also find concerns that must be addressed using SSM concepts. This case study demonstrates a general means for determining the applicability of SSM and how selection argumentation can be used to specify the general SSM mechanics within the argument. Engineers can then use the resulting assurance case as a specification for development of a fully implemented SSM process, thereby supporting practical application of SSM.

1.6 Dissertation Outline

The organization of chapters is as follows:

- Chapter 2 provides an overview of the rationale and notations used for assurance arguments, which serves as a fundamental driver for the SSM model.
- Chapter 3 defines the core concepts of the SSM model. A thorough description of the rationale, and general form and mechanics of the SSM model are described in terms of the SSM process

and the SSM assurance case. Subsequent chapters further address challenges associated with applying assurance cases to SSM.

- Chapter 4 defines a specific form for the SSM assurance case for examination and evaluation. Specifically, this chapter defines the structure and mechanics for developing a fitness argument for security. A fitness argument is one of two necessary arguments within an SSM assurance case.

- Chapter 5 defines a novel security metric based on the evaluation of argument confidence. This framework is presented to describe how confidence can be expressed within an assurance argument and how confidence arguments could be evaluateded to assess the quality of software (in this instance, security is used as a target quality for illustration).

- Chapter 6 defines novel extensions to traditional argument notation and mechanics to support the use of assurance cases for SSM.

- Chapters 7, 8, and 9 each present a case study evaluation of SSM. The details and purpose of each case study are described in the previous section.

- Chapters 10 and 11 discuss the related work and conclusions of this dissertation.

Chapter 2

Assurance Arguments

SSM makes extensive use of assurance arguments both as a method for justifying the adequacy of SOUP modifications and as a method for governing the mechanics of how modifications are generated. Before a further discussion of the concepts and mechanics of SSM are provided, this chapter presents an overview of traditional assurance argument technologies by describing:

- the general motivation and need for assurance arguments,
- a common method for documenting and structuring arguments, and
- how arguments are used to explain evidence and build assurance in software systems.

2.1 The Need for Assurance Arguments

When engineering software systems, developers are left with the burden of demonstrating assurance that the software establishes properties and characteristics desired by the system stakeholders. For example, stakeholders might require assurance that the software is adequately safe, secure, reliable, etc. for its use (i.e., for its operating context). A software system is said to be *acceptable* for its operating context if adequate assurance is demonstrated that the system has the stakeholder-desired property or properties.

The complexity and size of modern software, however, makes providing definitive, complete, and irrefutable proof that any given software system is acceptable impractical for all but the most basic and trivial software. In practice, developers rely on any available evidence to demonstrate that the existence of desired properties is "highly probable", although quantification of such probabilities is often impossible. Consequently, interpretation of what constitutes highly probable assurance is left to intuition.

Evidence alone does not provide a justification that a given software system is acceptable for its specific operating context. Rather, evidence must be explained and interpreted to demonstrate assurance. There are essentially two approaches for explaining evidence:

Prescriptive Standards: Evidence can be used to demonstrate compliance with a standard. A prescriptive standard specifies a set of required procedures and practices developers must follow and evidence developers must produce. A prescriptive standard can be conceptualized as a recipe that when followed is interpreted as demonstrating a software system is acceptable for its operating context.

Goal-Based Approaches: In a goal-based approach, evidence is used to support stakeholder-defined claims (in the form of goals) about the system's properties. Unlike prescriptive standards, the content and number of goals are not necessarily prescribed nor is the manner in which goals are justified by evidence. This freedom supports flexibility for developers to define assurance as they deem appropriate. A goal-based structure is interpreted as demonstrating adequate assurance that a software system has stakeholder-defined properties based on a thorough examination of the goal structure, the operating context, and the specific needs of the system stakeholders.

Prescriptive standards are often developed and reviewed by numerous experts and therefore capture a wealth of knowledge about building acceptable systems. Developers can use the standard as a checklist and simply provide the specified evidence. Consequently, prescriptive standards are

somewhat easier to understand and verify than goal-based approaches. The primary limitations of prescriptive standards are that:

- Standards are inflexible. The specific needs of stakeholders are not addressed, i.e., standards do not take into consideration the characteristics of a specific software system and its operating context.

- Standards rarely contain explicit rationales explaining why they demonstrate acceptable assurance, and therefore prohibit deeper understanding of the assurance any given standard is meant to provide.

- Standards rely upon the assumption that adherence to a given standard results in an acceptable system in all uses of the standard.

A goal-based approach to explaining evidence allows developers to overcome the limitations of prescriptive standards. In a goal-based approach, claims about the properties of a software system are made based on the needs specific to the system stakeholders. Evidence is then used to support the specified claims. The key benefit of this approach is that assurance is tailored to a specific software system. Consequently, developers must take a more active role in defining what evidence should be collected and rationalizing how that evidence supports claims. *Assurance arguments* are one method for explaining evidence using a goal-based approach.

An assurance argument is a structured decomposition of a general claim about a software system, referred to as a *top-level* goal or claim. The top-level goal of an argument specifies a general property the software system has and is essentially the conclusion that the argument is meant to support. The top-level goal is subdivided into sub-goals recursively. Eventually, a goal is no longer subdivided and supported directly by evidence. The premise of an assurance argument is that if the subdivision of goals (i.e., the argument structure) and evidence in support of goals is complete, trustworthy, and appropriate (i.e., valid), then the system has the property specified in the top-level goal.

2.2 Assurance Cases

In system safety, the construction of a rigorous *safety case* is gaining popularity. A safety case is defined as follows [17]:

Definition 2.1. ***Safety Case***: *A Safety Case consists of a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment.*

The concept underlying a safety case is the creation of a *compelling argument*, which justifies a claim that a system is adequately safe for its operating context. Since definitive proof that a system is adequately safe is impractical, the goal of the safety case is to show how all available evidence about a system supports claims that the system is adequately safe. Safety cases are required by law for certain military systems, including those used by the U.K. Ministry of Defence [17].

The safety case concept is not restricted to safety, and can be generalized and applied for the purposes of generating an explicit rationale for belief in any system characteristic of interest. This generalization of a safety case is referred to as an *assurance case*. In this dissertation, the illustrated application of SSM is for security modifications. Assurance cases are used to demonstrate the resulting modified software is adequately secure. Assurance cases adopted for security are commonly referred to as *security cases*.

2.3 Documenting Arguments

While in principle an assurance argument can be documented in any form, including natural language and tables [18], an increasingly popular and effective documentation method is to structure the argument using a graphical notation. In a graphical notation, components of the argument, such as claims and evidence, are represented as nodes in a graph. The connections/relationships between these nodes illustrate how evidence supports claims and thereby forms an assurance argument.

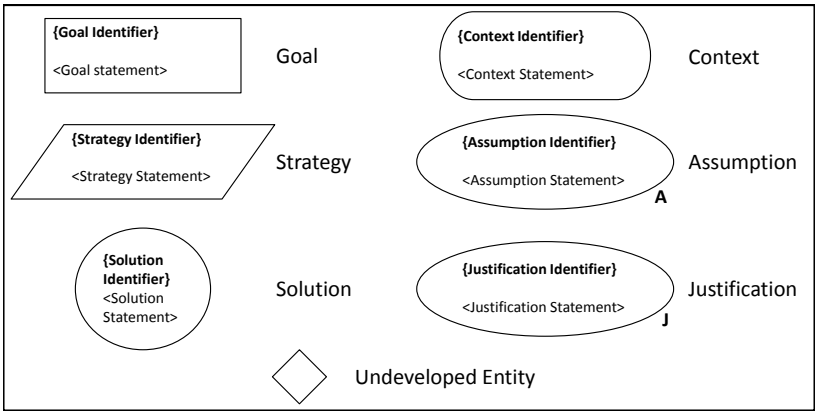


Figure 2.1: Core GSN elements

Currently, there are two commonly used graphical notations: the Goal Structuring Notation (GSN) [18, 19] and the Claims, Arguments and Evidence notation (CAE) [20, 21]. The distinction between these notations is not important in the scope of this dissertation, and only one notation is needed. As such, GSN is chosen as the notation of preference for the remainder of this work. The following subsections describe the core concepts of the GSN notation; however, a more complete and thorough description can be found in the GSN community standard [19].

2.3.1 Goal Structuring Notation (GSN)

The core GSN elements, shown in Figure 2.1, are:

Goals Depicted as a rectangle, a goal documents a claim about a property or a characteristic that a software system is said to have. Each assurance argument contains a top-level goal, which is the conclusion the argument is meant to support. The top-level goal is subdivided hierarchically into sub-goals. Sub-goals are refinements/simplifications of higher-level goals, and represent claims about a more specific sub-system or property of the larger software system.

Contexts Depicted as an oval, a context provides a reference to contextualizing information and documentation. For example, a context can refer to limitations about the scope of a goal. A

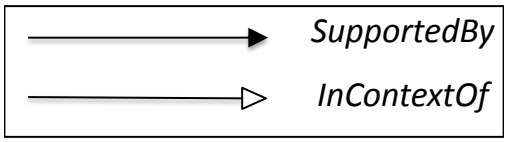


Figure 2.2: GSN relationships

context is linked to the argument element requiring contextualization, and all sub-arguments from that element inherit the context.

Assumptions Depicted as an oval with the letter 'A' at the bottom-right, an assumption is a specialized context element used to present intentionally unsubstantiated statements.

Justifications Depicted as an oval with the letter 'J' at the bottom-right, a justification is a specialized context element used to provide a rationale for a component of the argument.

Strategies Depicted as a parallelogram, an argument strategy describes the inference between a goal and its sub-goals.

Solutions Depicted as a circle, a solution is used to reference evidence in direct support of a goal.

Undeveloped Entities Depicted as a hollow diamond, the undeveloped entity symbol is directly placed on any of the above argument elements to indicate that the element is intentionally left undeveloped by argument engineers. Undeveloped argument entities signify further review and examination of the undeveloped element is necessary.

GSN argument elements are connected to each other through relationships depicted as arrows shown in Figure 2.2. The direction and style of the arrow indicates the type of relationship, and can take one of two forms:

***SupportedBy* Relationships** Depicted as a solid arrow, a *SupportedBy* relationship indicates inferential and evidential support, i.e., relationships between strategies, goals, and evidence. The arrow points to the supporting argument element. For example, evidence is connected to a goal by a *SupportedBy* relationship pointing to the supporting evidence (i.e., a solution element).

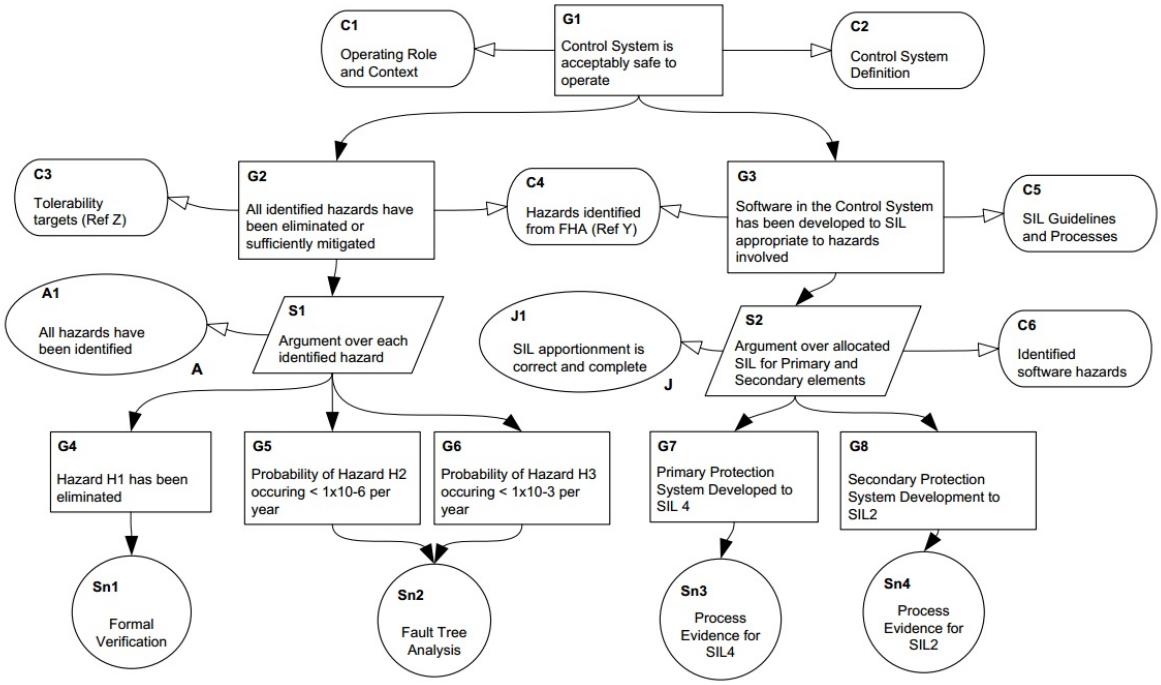


Figure 2.3: Example GSN argument fragment — figure taken from the GSN community standard [19]

***InContextOf* Relationships** Depicted as a hollow arrow, an *InContextOf* relationship indicates a contextualizing relationships where the arrow points to the argument element providing the contextualization, i.e., a context, a justification, or an assumption. For example, a goal that relies on an assumption uses an *InContextOf* relationship to point to the contextualizing assumption element.

Figure 2.3 illustrates an example argument structure using the GSN notation taken from the GSN community standard [19]. In this example, the top-level, labeled G1, is justified through a structured decomposition of sub-goals. Ultimately, the top-level goal is supported by four items of evidence (labeled Sn1, Sn2, Sn3, and Sn4). The argument structure shows how these items of evidence relate to and justify the top-level claim.

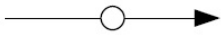
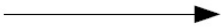
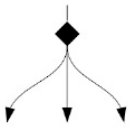
Multiplicity Extensions	
	A solid ball is the symbol for many (meaning zero or more). The label next to the ball indicates the cardinality of the relationship
	A hollow ball indicates "optional" (meaning zero or one)
	A line without multiplicity symbols indicates a one to one relationship (as in conventional GSN)
Optionality Extension	
	This symbol is defined for use over the existing relation types. Choice can be used to denote possible alternatives in satisfying a relationship. It can represent a 1-of-n and m-of-n selection.
Entity Abstraction Extensions	
 Uninstantiated Entity	Entity remains to be instantiated i.e. at some later stage the 'abstract' entity needs to be replaced (instantiated) with a more concrete instance.
 Undeveloped Entity	This placeholder denotes that the attached entity requires further development, i.e. at some later stage the entity needs to be (hierarchically) decomposed and further support by sub-entities.

Figure 2.4: GSN pattern notation elements — figure taken from Habli and Kelly [22]

2.3.2 GSN Pattern Extensions

Arguments vary between software systems, however, similar argument structures can be found in different arguments. Reuse of general argument patterns is common, therefore documenting these patterns facilitates easier argument development. Instead of having to create an argument with no guidance whatsoever, argument developers can consult published pattern libraries to find common methods of argument that have been previously studied and applied.

GSN supports the documentation of argument patterns through a set of pattern notation extensions, shown in Figure 2.4 (this figure is taken from Habli and Kelly [22]). These extensions allow for documenting general argument structures not tied to any particular implementation through abstractions about the argument structure (structural abstraction) and the entities referenced in argument elements (entity abstraction):

Structural Abstraction: Represented by the multiplicity and optionality extensions in Figure

2.4, these extensions express n-ary, optional, or alternative relationships between argument elements. Structural abstraction is used when the exact number or form of argument structures can vary depending on the specific use of the pattern. For example, an argument structure might repeat, but the number of repetitions depends on the context in which the argument is used. An n-ary relationship (a solid ball on an arrow) can be used to indicate that the argument will repeat an undetermined number of times.

Entity Abstraction: Represented by entity abstraction extensions in Figure 2.4, these extensions are attached to argument elements to indicate the element is either not instantiated or requires further development not described in the pattern. Entity abstraction allows generalization and specialization of argument elements. For example, a goal within a pattern can make a claim about a general property. The specific property depends on the instantiation of the pattern; hence, the uninstantiated entity notation would be connected to this goal to indicate developers must select a specific property.

When GSN abstractions are applied within the argument, the result is an argument pattern that can be instantiated for various uses. The pattern is instantiated by removing abstractions and replacing them with concrete structures or entities specific to a given software system.

An example argument pattern from the Assurance Based Development (ABD) pattern library [23] is shown in Figure 2.5. This pattern is called the "correct use of correct tool" pattern and is used to make an argument that a given *artifact* in the argument has some defined *property*. Note that all entities within curly braces, including the concept of an artifact and a property, are entity abstractions. Using this pattern requires instantiating these entities. The premise of this pattern is that if the tool used to generate the artifact is correct and the tool was used correctly, then the artifact would have the specified property.

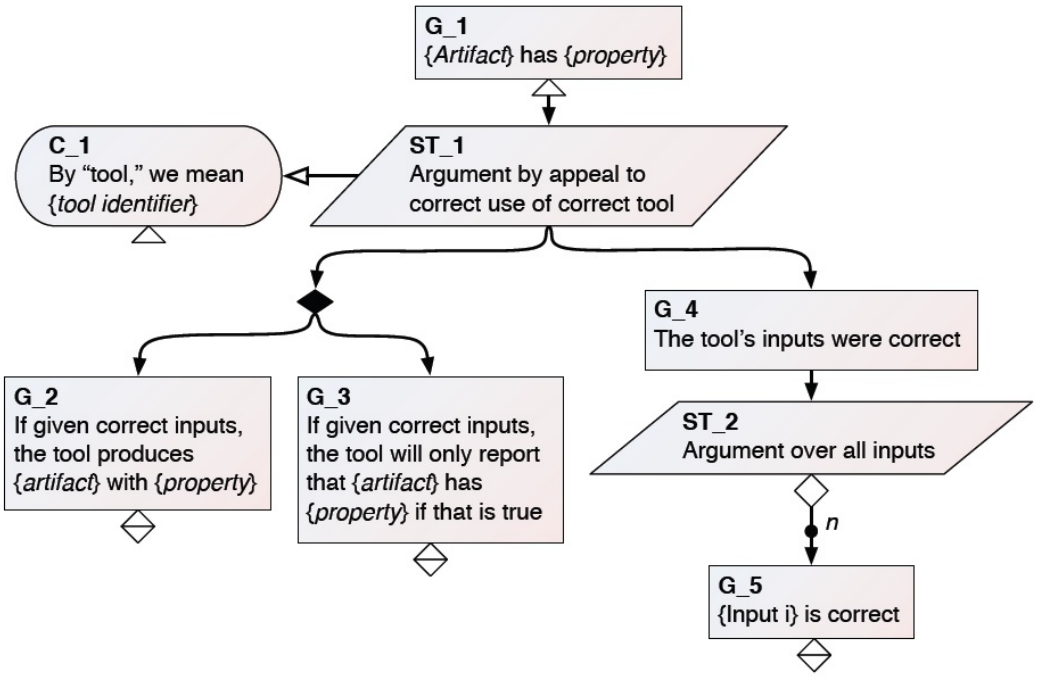


Figure 2.5: Correct use of correct tool argument pattern — figure taken from the Assurance Based Development (ABD) pattern library [23]

2.4 Understanding Argument

Using GSN, an argument is constructed as a graphical and hierarchical structure, where a top-level goal is subdivided recursively until a goal can be directly justified by available evidence, as illustrated in Figure 2.3. When a goal is subdivided, justified by evidence, or contextualized, an inference is made about the relationship between argument elements. Ideally, all inferences within an argument would be based on deductive reasoning. In deductive logic, if the premises are true, then the conclusion is necessarily true. In this manner, the argument could serve as a proof supporting the top-level goal. In practice, however, application of deductive logic in support of claims about real-world software systems is not always possible. Consequently, arguments about software system properties rely primarily on *inductive reasoning*.

In inductive logic, if the premises are true, the conclusion is "likely" true. The argument does not offer irrefutable proof that a top-level goal is valid. The likelihood that the top-level goal is valid is based on a careful and systematic examination of all risks and available evidence. Because

arguments must be constructed carefully and then thoroughly examined, these arguments are often referred to as *rigorous arguments*.

The use of inductive logic means assurance arguments are provisional and subject to revision as new information becomes available. An assurance argument is said to be *defeasible* since evidence can become available in the future refuting a top-level claim (such evidence is typically referred to as a *defeater*). The defeasibility of an assurance argument stems from two primary sources of doubt [24]:

Logical Inference Doubt: Doubts about the accuracy of the reasoning used in the argument, i.e., doubt that each step of logical inference follows to justify a top-level goal. These are doubts about the validity of relationships between argument elements. For example, to subdivide a goal into three sub-goals, there might be doubts that the subdivision is complete, i.e., perhaps more than three sub-goals are necessary. Similarly, the appropriateness of the subdivision might be questioned, i.e., do the sub-goals logically follow to provide justification of the goal.

Epistemic Doubt: Doubts about the completeness and accuracy of the knowledge about the software system. This knowledge takes the form of evidence, supporting documentation, and generally any information referenced within the argument. For example, evidence in the form of testing results might be questioned because the testing procedure might be flawed or those performing the tests are unqualified or untrustworthy. For example, the test results might be mishandled and consequently corrupted during and/or after testing. Any use of this evidence would therefore result in a misleading argument.

Some individuals might question the benefit and the precise purpose of assurance arguments if the argument is potentially defeasible: i.e., without definitive guarantees about the top-level goal, what use (if any) do assurance case technologies offer? Although an assurance argument cannot typically be used as proof about a top-level claim about a software system, the primary benefit of assurance arguments is the explicit documentation of reasoning and rationale for why a system is considered acceptable.

All software systems determined to be acceptable for use rely on some form of argument, even if that argument is implicit. By making the argument explicit, assurance arguments open up the software system for active scrutiny and criticism. The argument can be challenged and reviewed exhaustively, supporting a more structured approach for finding flaws/weaknesses in the software system (an example argument evaluation mechanism is defined in Chapter 5). As the system is updated, either in response to a found flaw or new functional needs, the argument is also updated, allowing developers and stakeholders to understand the impact of alterations and to determine if further changes to the software are necessary.

While an assurance argument provides exhaustive and transparent justification of a top-level goal, the complexity and size of the argument can make determining if the top-level goal is indeed adequately justified challenging. Although the explicit documentation of the argument allows for review and discovery of flaws in the argument itself [25], there are always residual doubts about whether all risks are mitigated adequately. The adequacy of the argument is context specific and is ultimately determined by the needs and opinions of the system stakeholders.

Chapter 3

Speculative Software Modification

This chapter defines the core concepts and components of Speculative Software Modification (SSM), an engineering model for generating SOUP modifications under uncertainty [26]. Modifications generated using the SSM model are intended to improve the dependability (e.g., safety or security) of SOUP, but successful application of a SOUP modification is often predicated upon unknown and uncertain data (see Chapter 1). To address the uncertainties of SOUP modification, the SSM model defines a process architecture for modifying SOUP to be instantiated by engineers.

The SSM process is defined as an abstract architecture. Engineers instantiate the architecture to meet the needs of specific system stakeholders. Conceptually, the instantiated process accepts an instance of SOUP as an input and produces a SOUP modification as an output. In this manner, an instantiated process can be reused to modify any given instance of SOUP thereby facilitating the modification of a large corpus of software.

To provide a rationale for why generated modifications are acceptable, the SSM model also mandates the construction of an *assurance case*. The SSM assurance case argues the acceptability of any modification produced by the SSM process. The SSM assurance case specifies and justifies the activities performed in the SSM process. The remainder of this chapter further describes the motivation, form and mechanics of the SSM process and assurance case.

3.1 Overview and Core Concepts

The SSM model is comprised of two fundamental and abstract components that are instantiated by engineers applying the model:

The SSM Process: A SOUP modification process where given an item of SOUP, the process generates a dependability-enhanced SOUP modification. The process, once instantiated, can be applied repeatedly to modify any provided SOUP.

The SSM Assurance Case: An argument supported by evidence providing an explicit rationale for why a SOUP modification produced by the SSM process is considered acceptable.

In principle, the process and assurance case have an interdependent and supporting relationship:

- the SSM process is a realization of the argument in the SSM assurance case, and
- the SSM assurance case provides an explicit rationale for the activities performed in the SSM process.

Figure 3.1 provides a high-level illustration of the SSM process and the interaction with the SSM assurance case. This section provides an overview of the goals and general mechanics of the SSM model and its components as illustrated in Figure 3.1.

3.1.1 SSM Process Overview

To address the uncertainties and variability associated with SOUP modification, the SSM process prescribes the development of *hypotheses* about how to apply SOUP modifications. A hypothesis is a supposition based on potentially unreliable information about how to effect a modification. Assessments are performed to validate hypotheses throughout the modification process. Hypotheses that fail assessment trigger the generation of new and refined hypotheses. Hypotheses passing all assessments (i.e., valid/acceptable hypotheses) are then used to effect a final modification. Iterative assessment and refinement occurs both within and between two primary sub-processes (referred to as phases), shown in Figure 3.1:

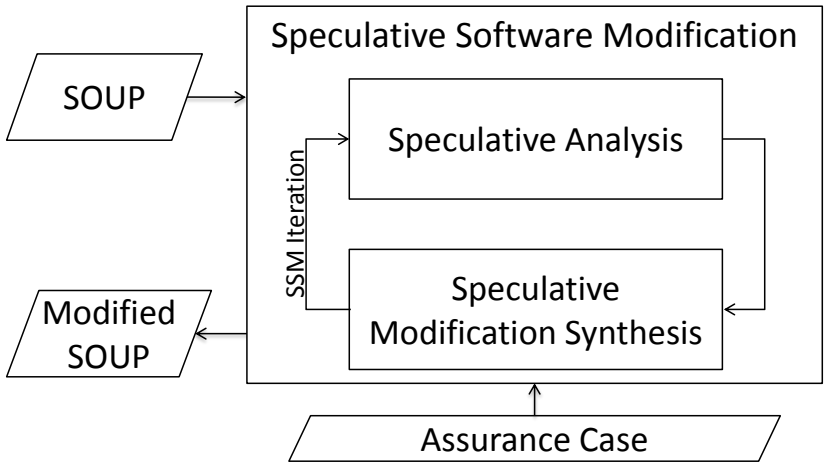


Figure 3.1: Abstract speculative software modification process architecture

Speculative analysis: The subject SOUP (i.e., the SOUP input) is analyzed to recover the information about the software necessary to design and apply a modification that enables the desired dependability properties. The recovered information might be unreliable and is therefore referred to as a hypothesis. Hypotheses are generated and then assessed to determine the likely quality of a modification that would result from their use. This process occurs iteratively until assessment has determined that a hypothesis has the potential to facilitate the desired modification (and hence the dependability properties of interest). Once a hypothesis is validated, speculative analysis is suspended.

Speculative modification synthesis: The hypothesis produced by speculative analysis is used to modify the software. Software modification can introduce new uncertainties affecting the quality of the modification, therefore necessitating modification assessment prior to deployment. Modification generation and assessment can occur iteratively if assessment reveals alternative modifications or configurations would be better applied. If assessment determines that the modification effects the property of interest, modification synthesis is terminated. If assessment reveals that the synthesized modification does not effect the property of interest with sufficient assurance (i.e., invalidates the hypothesis generated in speculative analysis), speculative analysis is resumed and the entire SSM process is repeated.

In generating and assessing the modified program (modification synthesis), assessment might reveal inadequacies about the hypothesis upon which the modification is based. As a result, speculative analysis is re-initiated in order to refine/reconfigure the hypothesis, referred to as *SSM iteration*. SSM iteration continues until either:

1. a modification is generated that is considered acceptable (acceptability is dictated in the SSM assurance case — see Section 3.1.2), or

2. no acceptable modifications can be produced.

Internally, each phase also performs iterative assessments and refinements (further described in Section 3.2). In this manner, modifications are tailored and customized based upon how hypotheses are validated. The SSM process shown in Figure 3.1 is used for a general discussion and overview. Consequently, the illustration intentionally belies much of the internal process complexity. A more detailed description of the internal mechanics of the SSM process is given in Section 3.2.

3.1.2 SSM Assurance Case Overview

An instantiation of the SSM process provides no explicit rationale for why generated modifications should be considered acceptable for use. Presumably, the SSM process is instantiated with care to meet the needs of the system stakeholders. If so, the process itself might imply an acceptability argument; however, implicit arguments and rationales are difficult to understand and review. To support an explicit model of modification acceptability, the SSM model mandates the construction of an *assurance case*. The SSM assurance case is implemented by engineers in addition to the SSM process.

Traditionally, an assurance case provides an argument that a single software system is acceptable for use. This approach, however, does not provide sufficient support for the SSM model. The SSM process can produce numerous and varying SOUP modifications. Consequently, the acceptability of each modification might be argued differently. To support variability in the argument, the SSM assurance case extends traditional argument techniques by providing:

1. the specification of all valid software analysis and modification options along with their acceptable configurations (referred to as the *acceptable solution space*),

2. the criteria for making decisions, i.e., generating hypotheses and navigating the acceptable solution space,

3. the criteria for validating decisions (i.e., for validating hypotheses), and

4. the argument justifying why a produced modification is acceptable.

By including how decisions are made and validated, the argument specifies the configuration and mechanics of the SSM process. Configuration information can then be used to alter the configuration (i.e., reconfigure) the process at run-time; therefore, the assurance case is depicted as an input to the SSM process in Figure 3.1. For example, the criteria for validating a hypothesis are documented in the assurance case. The validation criteria might be prone to alteration over time as the operating context and needs of the system stakeholders evolve. Instead of hard coding the criteria in the instantiation of the SSM process, the current validation criteria, as found in the assurance case, can be referenced dynamically. Use of the assurance case as an input for reconfiguration is at the discretion of SSM engineers.

Section 3.3 provides a more detailed description of the motivation and goals of the SSM assurance case. Section 3.3 also describes the challenges of applying assurance cases to support SSM. These challenges necessitate complex and detailed solutions thoroughly addressed in subsequent chapters.

3.1.3 SSM Goals

The goal of an instantiation of the SSM model is to produce SOUP modifications that establish stakeholder-defined dependability properties (e.g., security or safety) in the software. The motivation for using SSM and the target users of SSM are described in Chapter 1.

Modifications generated by the SSM process are intended to be *acceptable, practical* and *proactive* even though they were developed in the absence of reliable, yet critical, information:

Acceptable Intuition might suggest that a modification is "reasonable" in spite of any associated risks (see Chapter 1). Rather than rejecting what could be a useful modification because of uncertainty, SSM speculates on whether the proposed modification is acceptable for target stakeholders, i.e., whether there is sufficient assurance that the modified software will remain useful.

Practical A practical modification establishes properties addressing real concerns and does so in a manner that addresses the associated modification risks. SSM makes risk assessment explicit during development of modifications. Practical software modifications are also produced using available and/or easily developed methods that are cost effective to produce and use.

Proactive Modifications are made without necessarily witnessing an event suggesting the need for modification, i.e., they are proactive. For example, a modification to provide protections against a security threat could be applied in response to witnessing an attack on the system; however, security enhancements are best applied proactively, prior to an initial attack. Further, modifications are validated proactively, i.e., prior to releasing the modified software.

In principle, SSM is predicated on the premise that generally there is not one idealized method for modifying SOUP to establish dependability properties; rather, there are various acceptable solutions, forming an *acceptable solution space*. Through iterative assessment and refinement mechanisms (see Section 3.2), SSM allows stakeholders to perform a structured exploration of the acceptable solutions space for each piece of SOUP requiring modification.

Definition 3.1. ***Acceptable Solution Space***: *The space of modification solutions (modification variations) that will be considered acceptable/tolerable to the stakeholders.*

Modifications within the acceptable solution space might not provide the same benefits or consequences. As such, some solutions might be better than others. If preferable options are not possible, however, stakeholders will accept or tolerate other modification solutions so long as they are within the scope of the acceptable solution space.

While the ultimate goal of SSM is to produce acceptable (e.g., secure) software modifications, an instantiation of the model does not provide any guarantees that SOUP will always be modified successfully. The process can fail to find an acceptable method for modifying the software and instead produce an error message output. A failure to produce acceptable modifications can occur for a variety of reasons, further discussed in Section 3.2.

3.1.4 Instantiating the SSM Model

An instantiation of the SSM model is a defined process for modifying SOUP. An instantiated SSM process is intended to modify numerous pieces of SOUP based upon a predefined model of acceptable modifications (i.e., the assurance case). The repetition of applying modifications therefore makes the process amenable to automation. Whether or not the process is automated or applied manually is at the discretion of the system stakeholders.

In practice, the structure of an instantiation of the SSM model is determined by the inputs to the process:

1. the kinds of programs stakeholders wish to modify, and

2. the stakeholders' acceptability requirements defined in the assurance case.

The details of the instantiated SSM process structure might involve substantial internal complexity within each of the two SSM phases (i.e., speculative analysis and modification synthesis). Additionally, there might be more complex interactions between the phases than described in this chapter. The SSM model defines a set of general principles and guidelines, and should be considered flexible to meet any implementation-specific needs.

The remainder of this chapter delves deeper into the concepts and mechanics of the SSM process and the SSM assurance case.

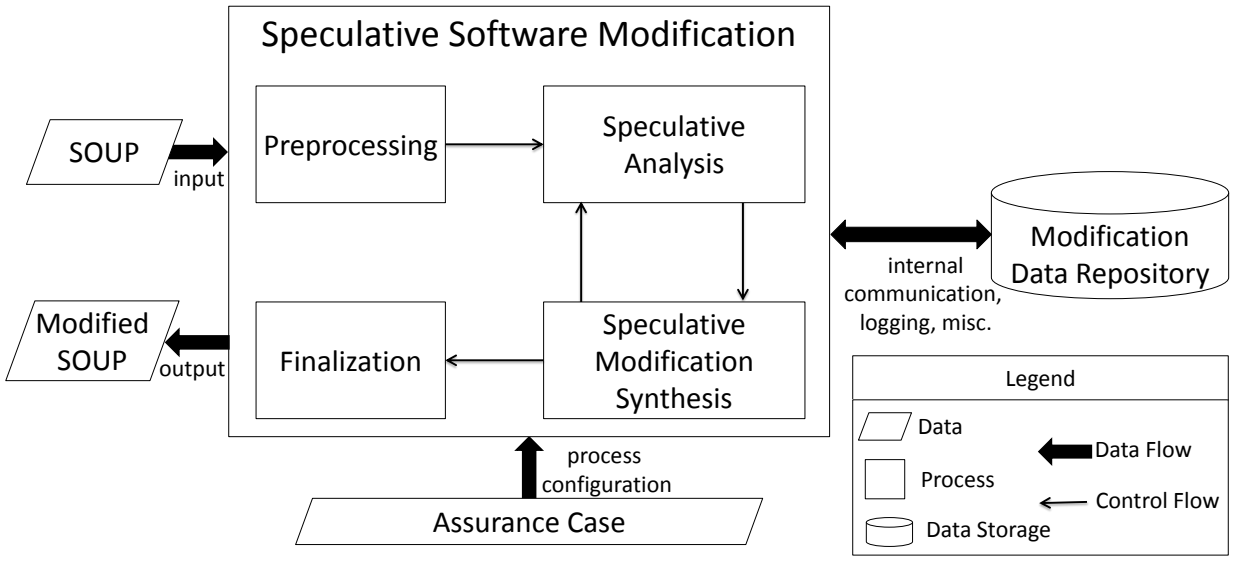


Figure 3.2: Detailed SSM process

3.2 The Detailed SSM Process

The detailed SSM process consists of four sub-processes (referred to as phases), illustrated in Figure 3.2:

- Preprocessing
- Speculative Analysis
- Speculative Modification Synthesis
- Finalization

Figure 3.2 describes the internal control flow between SSM phases when generating a SOUP modification. The figure also describes the flow of data into and out of the SSM process. SOUP enters the process and initiates the preprocessing phase. Control flow transitions through the process phases ultimately to the finalization phase where the modified SOUP is released. The SSM assurance case can be used to configure any phase within the process, and therefore is depicted as a process input for configuration purposes (further discussion of the purpose and mechanics of the assurance case is given in Section 3.3).

Each phase can access and record data that is later used or altered by subsequent phases or used for general bookkeeping, logging, etc. To support any kind of internal process communication, logging, storage/retrieval of analysis results, etc., that is not otherwise made explicit in the model, a common data storage mechanism is specified, i.e., the modification data repository. The manner with which the data repository is accessed is entirely dependent on the implementation of the SSM process and is therefore not prescribed in the model. The remainder of this section describes the detailed concepts and mechanics with respect to the internal SSM phases.

3.2.1 Preprocessing

The preprocessing phase is the first phase of the SSM process performed prior to generating and refining any SOUP modifications. The purpose of the preprocessing phase is to generate any data that will be useful for the remainder of the SSM process. Example preprocessing activities include:

Generating *baseline* information and statistics about the SOUP and its characteristics. For example, the SOUP disassembly might be referenced repeatedly by many different analyses in the SSM process. In this scenario, preprocessing can generate the SOUP disassembly in order to have it available prior to performing any analyses. Additionally, preprocessing can collect any relevant characteristics and statistics about the disassembly, such as the types of instructions, a subdivision of instructions per function, the libraries the software depends upon, and cyclomatic complexity [27].

Performing common or likely needed operations on the baseline data. Certain uses of the baseline data might require that the data undergo sanity checks, pattern filtering, normalization, etc. For example, disassembly of a binary program is often unsound [13], i.e., parts of the disassembly of the SOUP might be incorrect. Preprocessing can be used to attempt to locate and filter erroneous disassembly instructions or functions prior to any further modification generation.

Generating or retrieving test inputs. If dynamic analyses are used within the SSM process, preprocessing can be used to automatically generate these inputs (e.g., by means of concolic test input generation [28]), or retrieve inputs from pre-existing databases.

Verifying the SSM process is applicable. The principles of SSM allow for variations in how a modification is applied, but certain key assumptions might be necessary for *any* modification to be successfully generated. If these assumptions can be shown to be invalid in preprocessing, the entire SSM process can be terminated and an error message generated without wasting any additional resources in modification generation.

Preprocessing essentially allows for the optimization of the SSM process by producing any crucial data that can be generated prior to modification generation. Because the detailed process mechanics of preprocessing are highly dependent on the particular instantiation of SSM, no further sub-processes are described. Information generated by preprocessing is made available to all SSM phases through the modification data repository. Once preprocessing has completed, the speculative analysis phase is initiated.

3.2.2 Speculative Analysis

Any modification designed to enhance SOUP with some crucial property begins with development of a concept of how the property might be effected. For example, for a security-critical application, a concept might be to identify the relevant software vulnerabilities and develop a modification that either eliminates the vulnerabilities or makes them inaccessible. For vulnerabilities that might lead to buffer-overflow attacks, the concept might be to intercept all buffer write instructions and check them before they are executed. No matter what the concept is, if it is to be turned into a practical modification of the subject software, then the information about the program upon which the concept relies has to be determined. Determining this information is the role of speculative analysis.

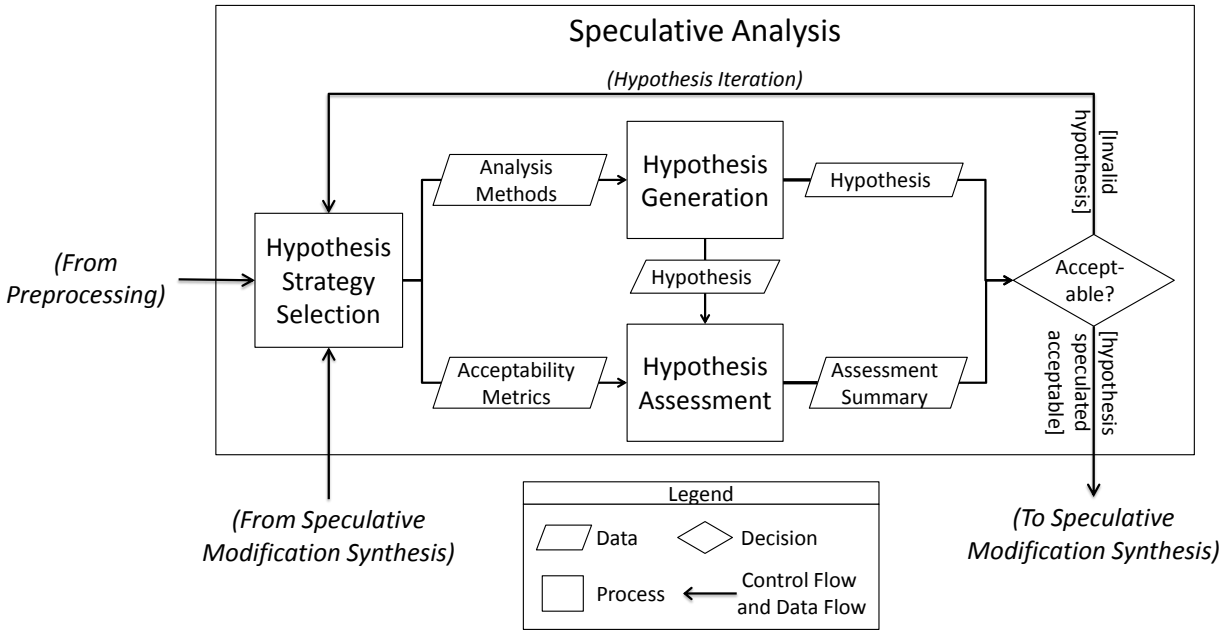


Figure 3.3: Detailed speculative analysis phase

The analysis is termed "*speculative*" because the information returned from a set of analyses might be incomplete or inaccurate (see Chapter 1). The rationale of speculative analysis is that chosen modifications, their configurations, and recovered data about the program might be deemed to be either "acceptable" or "unacceptable" as determined by the stakeholders. The potentially unacceptable data is treated as a hypothesis that is assessed, and if needed, refined, reconfigured, or discarded. The iterative generation and assessment of hypotheses within speculative analysis is referred to as *hypothesis iteration*.

The detailed internal process of the speculative analysis phase is illustrated in Figure 3.3. This figure describes both the control flow and data flow within the speculative analysis phase. The data flow within the phase describes key data structures generated and consumed by sub-phases; however, generally all sub-phases can access any necessary data through the modification data repository (see Figure 3.2). The following subsections describe the mechanics of the speculative analysis sub-phases in more detail.

Hypothesis Strategy Selection

The purpose of the first sub-phase of speculative analysis (hypothesis strategy selection) is to select a strategy for generating and assessing a hypothesis, referred to as a *hypothesis strategy*. A hypothesis strategy consists of:

1. methods for analyzing the software to produce a hypothesis, and
2. metrics for assessing the corresponding software analysis results (i.e., the generated hypothesis).

Implicit in the strategy is a set of modification techniques that can use the hypothesis to effect the desired dependability properties. The manner by which an initial strategy is selected is based on preferences specified by the stakeholders. Hypothesis strategy selection is re-entered if a hypothesis is later invalidated through assessment. The nature of the invalidation is used to govern selection of the next hypothesis strategy.

Hypothesis Generation

Hypothesis generation applies (i.e., performs) the set of analyses selected in hypothesis strategy selection. Any data necessary to perform the set of analyses (e.g., the SOUP, the SOUP disassembly, or test inputs) are accessed through the modification data repository. By performing the selected analyses, a hypothesis is generated.

The hypothesis can be assessed *directly* or *indirectly*. If assessed directly, the hypothesis is sent to the hypothesis assessment sub-phase. In some cases, hypothesis assessment can be performed on a partially completed hypothesis. In this scenario, partial hypotheses can be sent to the assessment sub-phase in sequence and assessed in parallel to hypothesis generation. If the hypothesis is assessed indirectly, assessment does not require access to the hypothesis (discussed in the following subsection).

Hypothesis Assessment

The hypothesis assessment sub-phase determines the utility of a generated hypothesis based on either direct assessment of the hypothesis or an indirect assessment through assessment of the SOUP itself. Note that in the latter scenario, assessment does not directly evaluate the hypothesis to determine its utility.

For example, an analysis might depend on key assumptions about the characteristics of the SOUP. Failure to have these characteristics will produce unreliable analysis results (i.e., an invalid hypothesis). Assessment can be used in this case to validate that these assumptions are met. If so, the hypothesis might be vetted without direct assessment, i.e., the analysis results are considered valid if the assumptions about the applicability of the analyses are valid.

Assessment computes quality metrics about the hypothesis and compares the results to acceptability requirements. The acceptability requirements are specified by the system stakeholders and are documented within the SSM assurance case. These requirements can be hard coded within the assessment sub-phase or the assurance case can be consulted dynamically.

Hypothesis assessment produces a report summarizing the acceptability of the produced hypothesis. If the hypothesis is determined to be unacceptable, the hypothesis strategy selection sub-phase is re-initiated. The hypothesis and the assessment summary are used to reconfigure the next strategy selection. This iteration is referred to as *hypothesis iteration*. If the hypothesis is validated, it is speculated to be acceptable. Speculative analysis is then suspended and the assessment summary and hypothesis are provided as input to the next phase of SSM, speculative modification synthesis.

3.2.3 Speculative Modification Synthesis

Modifications are generated, i.e., synthesized, once speculative analysis generates an acceptable hypothesis. In essence, modification synthesis is the practical application of the hypothesis to generate a modified program. Modification synthesis is also termed “speculative” because generating a modification might introduce new doubts. Additionally, the modification is built upon a hypothesis, which might be invalidated by assessment of the modification.

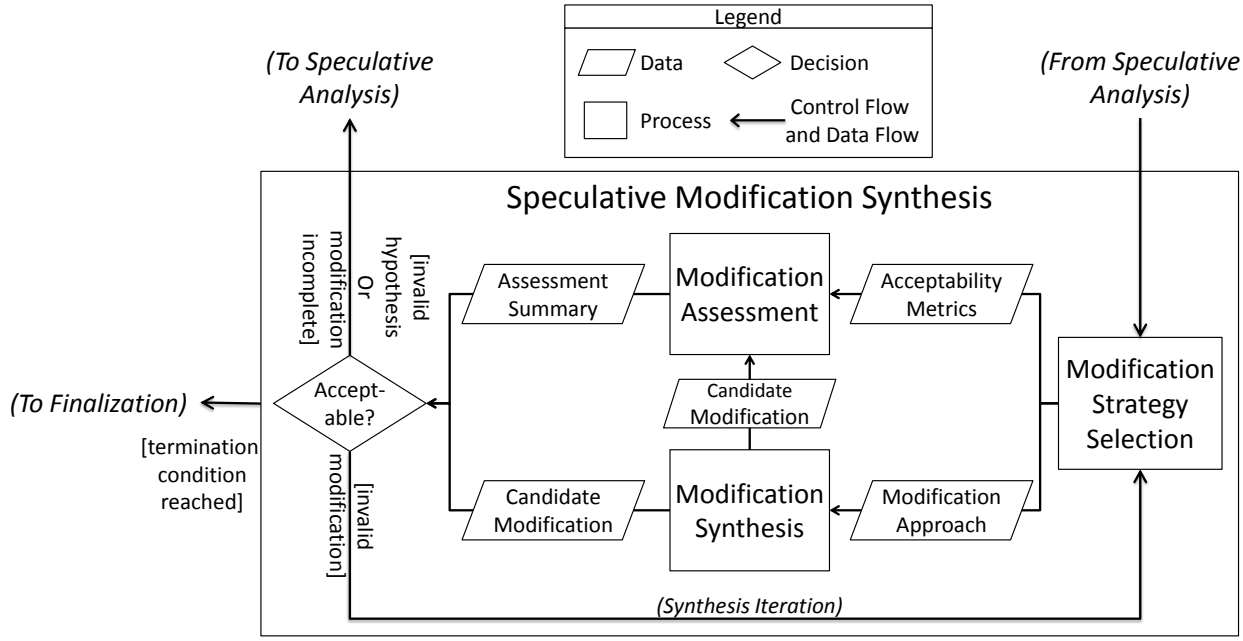


Figure 3.4: Detailed speculative modification synthesis phase

The detailed internal process of the speculative modification synthesis phase is illustrated in Figure 3.4. The internal process shown in Figure 3.4 describes both the control flow and data flow of key data structures within the speculative modification synthesis phase. Any data required by sub-phases but not explicitly provided as an input are accessible through the modification data repository. The following subsections describe the sub-phases of speculative modification synthesis in more detail.

Modification Strategy Selection

The first sub-phase, modification strategy selection, reviews the hypothesis and assessment summary produced by speculative analysis in order to determine how a modification should be applied and assessed. In principle, a hypothesis might be usable for multiple modification approaches. The purpose of modification strategy selection is:

1. to choose a specific approach for using the hypothesis to generate a modification, and

2. to choose how a produced modification is assessed.

The chosen modification approach and modification assessment metrics are referred to as a *modification strategy*.

Because multiple modification approaches can use the same hypothesis, iteration might be required to find the most applicable modification approach. In this scenario, modification assessment invalidates the modification approach but not the general hypothesis upon which the modification is based. Modification strategy selection is then re-initiated and a new strategy is selected based upon how the modification was invalidated.

Modification Generation

The modification generation sub-phase applies the selected modification approach. Modification generation accesses any data required to effect the modification (e.g., the SOUP, and the hypothesis produced by speculative analysis) through the modification data repository. The produced modification is referred to as a *candidate modification*.

A candidate modification can be assessed either directly or indirectly. If directly assessed, the candidate modification is sent to the modification assessment sub-phase. Indirect assessments, however, do not require access to the candidate modification. Indirect assessment of a candidate modification is described in the next subsection.

Modification Assessment

Modification assessment uses chosen assessment metrics (e.g., static or dynamic analyses) to either directly assess a candidate modification or to assess a candidate modification indirectly. One approach we propose for indirect modification assessment is the use of an *error amplification*. An error amplification is an exaggerated/distorted modification designed to increase the visibility of errors (i.e., to amplify errors) when the modification is assessed. An "error" in this context refers to any properties of a modification that could render the modification ineffective, incorrect, inefficient, or, more generally, unacceptable in any way (i.e., a user would be in *error* to apply the modification). Error amplifications are not intended for final use and are therefore discarded after assessment.

For example, if stakeholders are concerned about run-time and memory efficiency, a modification can be generated that intentionally consumes large amounts of CPU and memory resources. The intuition is the desired modification would not consume as many resources as the error amplified modification. The error amplification is therefore intended to act as a worst case scenario for resource consumption. If, when assessed, the error amplified modification is determined to be adequately efficient, then stakeholders might consider the intended modification adequately efficient.

Definition 3.2. ***Error Amplification***: *An error amplification is a software modification developed with the intent of determining whether a modification approach will be successful. An error amplification is only meant for assessment purposes. Hypotheses about how to modify software might have errors. An error amplification increases the visibility of potential errors (i.e., amplifies errors) through a modification that either (1) exaggerates characteristics of the software, (2) exaggerates characteristics of the hypothesis or modification approach, or (3) adds extensive instrumentation.*

Metrics computed by modification assessment are compared with acceptability requirements in the assurance case, and an assessment summary is produced documenting assessment results. The assessment summary can report that the candidate modification is invalid in two ways:

1. assessment has revealed new evidence invalidating the hypothesis generated by speculative analysis, or

2. the hypothesis is still valid, but the modification approach itself was invalidated.

In the former case, speculative analysis is re-entered to produce another hypothesis based on the results of assessment, referred to as SSM iteration. In the latter case, the assessment results are used to reconfigure modification strategy selection, referred to as *synthesis iteration*. Synthesis iteration is intended for scenarios where the hypothesis can be reused by different modification approaches. For example, if assessment determines that the run-time overhead exceeds a specified threshold, the modification can be reconfigured and applied less extensively to increase efficiency. These changes in the modification do not require generating a completely new hypothesis.

Modifications can be performed in SSM on the entire program at once, or modifications can be applied to components of the program in a specified order. If the candidate modification is acceptable, the modification is not released until all modifications and components of the SOUP have been processed. SSM iteration continues for the next modification or component until a termination condition is reached (see Section 3.2.4).

3.2.4 Termination of SSM Iteration

Iteration within the SSM process continues until a *termination condition* is reached. Implicitly, iteration will terminate when all components of the software have been modified or no additional hypotheses can be generated. These conditions are referred to as the *implicit termination conditions*.

Definition 3.3. ***Implicit Termination Conditions***: *Termination of the SSM process due to arriving at an acceptable modification or having exhausted all acceptable options for producing a modification.*

Other *explicit* termination conditions can be specified at the discretion of the system stakeholders and the SSM engineers. Explicit termination conditions are primarily intended to restrict the resources consumed when generating modifications (e.g., time and memory); however, other uses of explicit termination conditions might be possible. Generally, SSM does not restrict where, for what purpose, how, and how often these additional termination checks can be applied.

Definition 3.4. ***Explicit Termination Conditions***: *Termination of the SSM process for reasons not defined in the implicit termination conditions. Uses of explicit termination conditions are at the discretion of engineers implementing the SSM process.*

3.2.5 Finalization

When a termination condition is reached, whatever modification has been produced up to that point represents a *best effort* modification, i.e., the best modification that could be produced with the

instantiated SSM process. Thus, modifications might be minimally applied, or, in the worst case, not applied at all.

Some stakeholders might require additional assessments to determine if the best effort modification is acceptable for use. Additionally, modifications can be produced and assessed at a component level (e.g., function-by-function). Consequently, termination of the process necessitates combining all modified components and assessing the composite modification. A modification finalization phase is defined in the SSM model, both to package the complete modification into a single modified program and to perform any final assessments.

Conceptually, assessments performed in the finalization phase are used to determine if the acceptability of the modified software can be argued adequately. The SSM assurance case specifies how an acceptability argument is constructed for SOUP modifications. Finalization verifies that all necessary evidence is present to support the argument in the SSM assurance case. If the modification cannot be supported by an argument, then finalization will report failure. Otherwise, the modified software is released for use. The details of the SSM assurance case are further discussed in Section 3.3

3.2.6 SSM Outputs

Ideally, the SSM process would always produce modified SOUP; however, the SSM process makes no guarantees that an acceptable modification will be produced. For example, no acceptable modification may be found, or the modification is rejected during assessments performed in the preprocessing or finalization phases. This leads to three possible outputs of SSM:

1. An acceptable SOUP modification.

2. An error message indicating no acceptable modification could be produced given the current SSM instantiation.

3. A warning message indicating further manual review is necessary to determine if the modified software is acceptable.

In the second case, the process has reached a conclusion that no modification can be generated given the acceptability requirements specified in the SSM assurance case. In the third case, the acceptability of the modification cannot be demonstrated nor invalidated. In both the second and third scenarios, a manual review is performed on the SSM instantiation and the conditions leading to the error or warning message. The process and the assurance case are then repaired/amended as deemed necessary by the SSM engineers. Maintenance activities performed on the SSM process and assurance case are out of the scope of this dissertation and not discussed further.

3.2.7 Engineering Considerations

The SSM process architecture is conceptualized as a linear process for the purposes of explanation; however, an instantiation of the process is not limited to linear execution. The process can be parallelized using multi-threading, cloud services, etc. to improve the speed by which SSM produces modified SOUP. For example, a MapReduce [29] paradigm can be used at the discretion of the engineers implementing the SSM process. Map and reduce operations can be performed on the entire process, individual phases, or internally to each phase. Mapping operations would essentially choose numerous hypothesis and modification strategies at once, and reducing operations would perform a post mortem assessment of the resulting acceptable approaches. Frameworks for the parallel execution of the SSM process are out of the scope of this dissertation, and not discussed further.

The process also involves some complex implementation considerations that are not specified by the model. Consider the assessment sub-phases in both speculative analysis and speculative modification synthesis. Based on the type of assessment, assessments can be performed before, during or after a hypothesis or candidate modification is produced. The order by which an assessment occurs depends on the kind of assessment activity and therefore can differ for each iteration. Similarly, assessment might not be necessary and omitted in either speculative analysis or modification synthesis for some iterations.

The presented SSM process is intended to provide general guidelines and principles for instantiating a process for SOUP modification. More complex interactions between phases and sub-phases are dependent on how the process is engineered and the needs of the system stakeholders; hence, these interactions are not specified within the model, and are left to the discretion of SSM engineers.

3.3 The SSM Assurance Case

Ostensibly, users applying any software modification have some form of rationale that the approach achieves some technical goal. This rationale can be ad hoc, even implicit, when all necessary information is available and the modification is based on appropriate analyses. When uncertainties remain about the software, an informal or ad hoc acceptability rationale is not sufficient. Risk remains when using the software, and that risk must be properly managed.

Uncertainties about the software force stakeholders to make implementation decisions about the modification that might affect effectiveness, correctness, efficiency, etc. in an effort to balance risks with potential advantages. The more decisions are made to distribute modification risks, the more complex and customized the acceptability rationale becomes. Consequently, implicit rationales become difficult and impractical to understand necessitating an explicit, systematic and structured rationale.

The structure used by SSM to model acceptability is an assurance case. The SSM assurance case uses expanded argumentation techniques (described in Chapter 6) to provide the following support for SSM:

1. A specification of the acceptable solution space (i.e., all acceptable modifications SSM can produce).

2. The criteria by which the acceptable solution space is to be explored (i.e., the techniques and configurations by which the SOUP is analyzed, modified, and assessed).

3. An argument as to why stakeholders should believe a produced modification is acceptable.

While the SSM process mechanically provides a means to search the acceptable solution space, the SSM assurance case explicitly governs the search and documents why a produced modification is within the solution space. The SSM assurance case is conceptualized as an input to the SSM process (see Figure 3.2), used to configure all decisions, analyses, modifications, evaluations, etc.; however, whether or not these decisions are configured dynamically from the assurance case is an implementation decision. The key relationship of the assurance case to the SSM process is that all decisions made in the SSM process are specified in the assurance case.

This section describes the motivation, goals, and challenges of applying assurance cases to SSM. The described challenges are then addressed in subsequent chapters.

3.3.1 Goal-Based Assurance

When a modification is produced by SSM, a reasonable user would naturally question its acceptability: e.g., is the modification adequately effective[1], efficient and correct? Achieving an ideal or perfectly acceptable modification is rarely possible, necessitating trade-offs in terms of the benefits and consequences a modification provides. SSM is intended for balancing these trade-offs.

The concept of an acceptable software modification is highly subjective. What is considered an acceptable modification is entirely dependent upon the context of its use and the needs of the system stakeholders. For example, one organization might be willing to accept considerable risks of having incorrect and highly inefficient software in order to have effective security defenses. Other organizations, however, might value correctness and efficiency over security properties. A model of modification acceptability, therefore, cannot be based on any prescriptive standard, since no universally accepted standard exists. Instead, acceptability must be based on a set of goals defined by the system stakeholders.

In a goal-based acceptability model, stakeholders define goals applicable for their context (i.e., what characteristics the modified must have for their operating environment) and demonstrate that

[1]With respect to our focus of applying SSM to enhance software security, an effective modification is one that provides the desired security benefit, i.e., adequately mitigates a defined set of attacks.

these goals are satisfied. There are two abstract approaches for applying goal-based acceptability models for SSM:

Product-Driven Goals: Stakeholders define goals about the characteristics the product of SSM (the modified software) must have to be considered acceptable.

Process-Driven Goals: Stakeholders define goals about the characteristics the SSM process must have in order to generate an acceptable modification.

In a product-driven approach, goals directly support the claim that a given software modification meets the stakeholders' concept of acceptability. In a process-driven approach, the acceptability of a modification is implied by the acceptability of the process: i.e., the process is shown to meet its acceptability goals implying anything the process produces is also acceptable.

Fundamentally, there is some overlap between the product (the modified software) and the processes used to generate it (the SSM process); however, the questions typically posed by stakeholders are centered on the quality of the modified software itself. Arguments should be explicit about what the modified software provides, not implicit (or indirect) through arguments about its construction. Therefore, the approach put forward in SSM is to model acceptability using product-driven goals.

To support a product-driven goal approach to SSM assurance, stakeholders require a comprehensive view of the benefits (and weaknesses) of a generated software modification. More specifically, stakeholders require support for structured reasoning, discussion, and analysis to allow for:

- development of a basic rationale for belief in a dependability (e.g., security) claim,
- development of confidence in that belief, and
- determination of whether the planned modifications should be applied.

Assurance-argument technologies (described in Chapter 2) provide much of this support; hence, our approach is to exploit the use of assurance cases to act as the acceptability model for SSM. In particular, assurance cases are used to:

1. Ascertain the adequacy of modifications in terms of the desired dependability properties.
2. Identify modification weaknesses and benefits.
3. Identify trade-offs in modification configurations.
4. Provide a structure that can be reviewed, searched, and updated, thereby allowing others to consider using the modified software or building on to the modification approach.

3.3.2 SSM Assurance Case Form

The general SSM assurance case form is predicated on two types of risks that an instantiated SSM process balances and mitigates when generating SOUP modifications:

Operational Risks: Risks corresponding to using (i.e., executing) a generated software modification, e.g., risks to modification effectiveness, correctness, and efficiency.

Developmental Risks: Risks corresponding to the development resources used in the generation of a SOUP modification, e.g., risks to the modification development time, memory consumption and budget.

SSM assurance cases therefore consist of two separate arguments to justify that each risk is adequately mitigated, as illustrated in Figure 3.5:

The Fitness (or Fit-For-Use) Argument: The fit-for-use argument, also referred to as a fitness argument, demonstrates that operational risks are mitigated for any given SOUP modification generated by SSM. A SOUP modification is fit for use if it adequately establishes desired dependability properties while balancing stakeholder-defined fitness constraints.

The Success Argument: The success argument demonstrates that developmental risks are mitigated when the SSM process generates a SOUP modification. The success argument demonstrates how an instantiated SSM process will be successful in producing a SOUP modification using acceptable developmental resources (e.g., time and money).

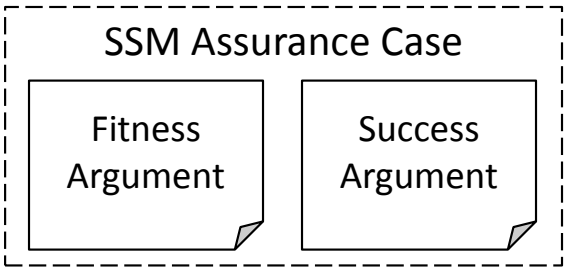


Figure 3.5: The general SSM assurance case form

The fit-for-use argument demonstrates the acceptability of generated modifications. Since the SSM model allows for variability in how acceptable modifications are generated, this argument must express a range of modification variability as well as the conditions under which these variations are restricted (i.e., the argument must document the acceptable solution space and how to navigate it). The success argument, however, justifies that the instantiated SSM process will use acceptable developmental resources. Restrictions on development resources act as a competing constraint affecting variability in the fit-for-use argument, but there is only one SSM process; hence, there is no variability in the success argument.

In essence, the success argument is based on claims and evidence about an implemented SSM process. Because the SSM process is in the control of the developers, all necessary evidence is available once the process is implemented. The fit-for-use argument, however, is based largely on claims and evidence about specific instances of SOUP. Such data is often not be available prior to modifying the software or developing the SSM assurance case.

3.3.3 Assurance Challenges

The concepts underlying the SSM assurance case presents significant research challenges. These challenges are motivated in the following subsections and discussed in further detail in subsequent chapters.

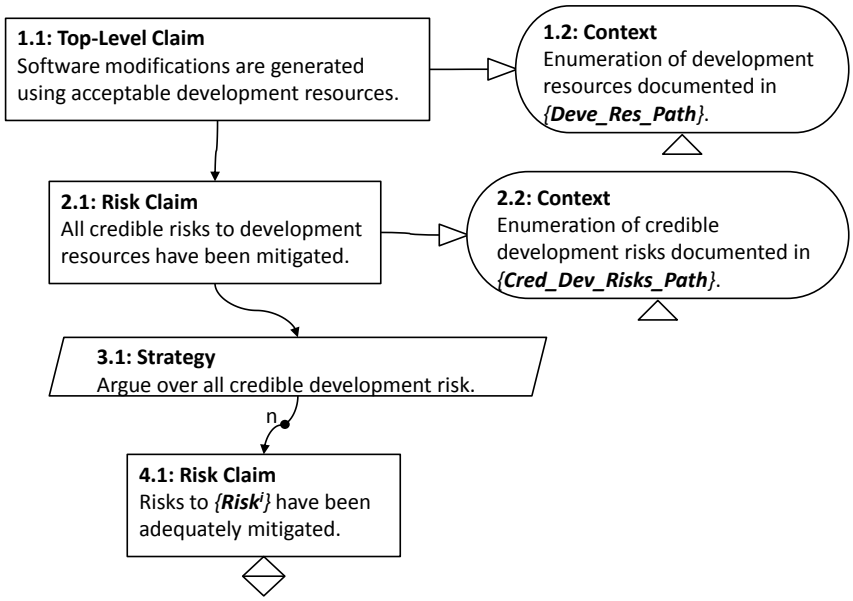


Figure 3.6: Example success argument pattern

Challenge 1: The Fit-For-Use Argument Form

The fit-for-use and success argument within the SSM assurance case are concepts originally proposed in Assurance Based Development (ABD) [23, 30, 31]. The ABD pattern library [23] provides suggested argument patterns for the construction of both the success and fitness arguments. Adaptation of the success argument patterns for SSM is relatively simple and is illustrated in Figure 3.6. The fit-for-use argument, however, depends largely on the kinds of dependability properties stakeholders wish to establish, and can become far more complicated than the success argument. In this dissertation, SSM is illustrated as a means for improving SOUP security; hence, Chapter 4 presents a detailed description of the form and rationale of a fit-for-use argument for security modifications.

Challenge 2: Arguing Confidence

The assessments within the SSM process can be conceptualized as generating/validating evidence within the SSM assurance case. Such evidence is used to support that:

- A given modification is effective (e.g., secure).

- A given modification will be effective within defined operational constraints, such as correctness and efficiency.
- A given modification can be generated within the developmental constraints specified within the success argument.
- Sufficient *confidence* exists that a given modification is fit for use.

Of particular interest is the latter issue of confidence: what exactly does confidence mean, how can confidence be expressed within an SSM assurance case, and what is the purpose of arguing confidence?

For simplicity, presented argument structures throughout this dissertation omit arguments typically associated with confidence (i.e., arguments of trustworthiness, appropriateness, and sufficiency). For example, the success argument structure in Figure 3.6 does not consider doubts about whether all development risks have been identified. Instead, confidence arguments are assumed to be present throughout the argument in separate confidence argument structures. In particular, we adopt and extend the concept of confidence arguments originally proposed by Hawkins et al. [32].

Fundamentally, the purpose of arguing confidence is to justify the quality of an argument, and by extension, the quality of the associated software system. A discussion of how to address argument confidence necessitates a discussion of how confidence can be used to measure software quality. Chapter 5 describes how confidence arguments can be constructed in the context of a novel argument metric framework. The metric framework is presented as a general concept, agnostic to the concepts of SSM; however, some potential uses for SSM are also discussed.

Challenge 3: Extending Assurance Argumentation

Traditional assurance argument notation methods are somewhat limited in providing the necessary support for SSM. Specifically, the purpose of the SSM fitness argument is to justify *any* generated modification is fit for use, not simply a single software system. Modification fitness can vary

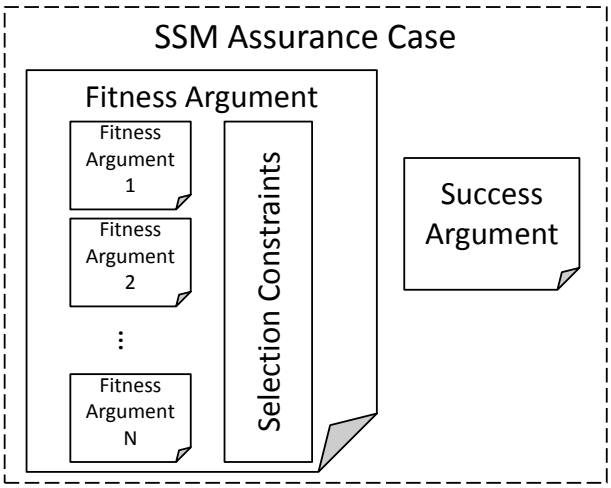


Figure 3.7: Detailed SSM assurance case concept

depending on the assessment and refinement mechanics of the SSM process; therefore, in principle, each modification produced by an SSM process will be justified by an individualized fitness argument.

The challenge of the SSM fitness argument is to express the range of all individualized fitness arguments that are possible in addition to the constraints governing how an individualized fitness argument is selected, as illustrated in Figure 3.7. Chapter 6 presents the structure and mechanics of *selection argumentation*, a set of argument notation concepts and techniques to allow the SSM fitness argument to express both the acceptable solution space of modifications (in this context the space of arguments that can be selected), and the criteria by which the solution space is navigated.

Chapter 4

Fitness Argument Structure for Security Modifications

> *"If we cannot understand the system well enough to build an assurance case, then we are not in a position to say that it is secure."* — Alexander et al., *Security Assurance Cases: Motivation and the State of the Art* [33]

Software modifications for security, either currently in use or in academic publications, are typically accompanied by some form of rationale and evidence to suggest that the given approach is useful. Rationales and evidence can be based on expert opinion, intuition, experimental results, mathematical proofs, etc. While this information might provide a general justification that a given software modification method has some utility, this justification is often informal, generalized and/or ad hoc.

Users lack specific justification for their particular operating context. Consequently, users wishing to apply these modification methods themselves require a reasoning structure for determining how a given modification method is useful for their specific goals and how the method should be applied. To provide an explicit and rigorous rationale for a set of specific system stakeholders, SSM mandates the construction of an argument in the form of an assurance case. Assurance cases used to argue system security are referred to as *security cases*.

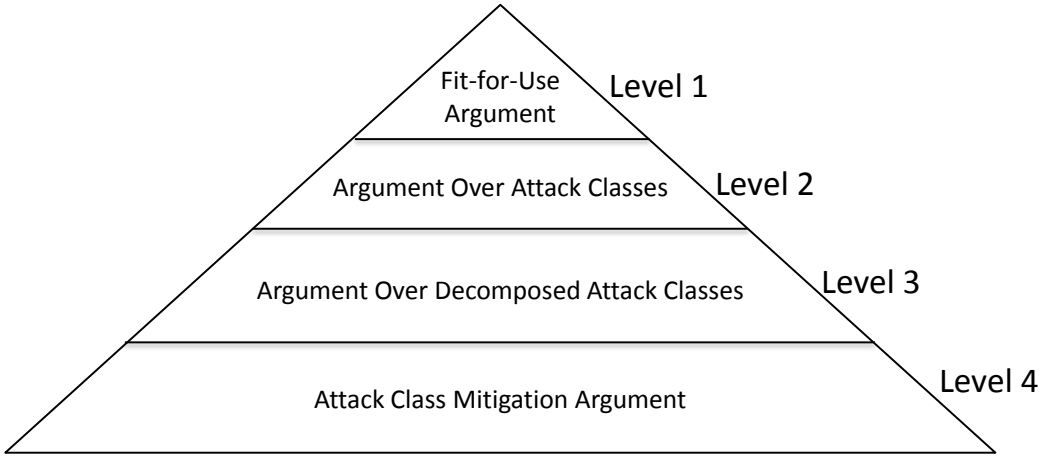


Figure 4.1: Abstract argument structure for security-enhancing software modifications

While uses of assurance cases for safety (safety cases) have been studied extensively and used over the past 20 years, applications for security have been explored minimally [33]. Consequently, applying security cases to argue the acceptability of security-enhancing software modifications, irrespective of SSM principles (i.e., iterative strategy selection, assessment, and refinement), is fundamentally challenging. Specifically, there are no widely accepted argument structures and guidelines for argument construction.

In order to facilitate the practical application of the SSM model for security modifications, this chapter presents a general argument structure for security-enhancing software modifications. With a general argument structure established, it can then be applied within the SSM assurance case; however, the structure is not limited to support SSM only. The argument structure might be applicable for other uses, and is therefore described for general purpose.

4.1 Structure Overview and Rationale

The purpose of modifying software for security is to mitigate possible attacks against the confidentiality, integrity and/or availability of valued assets. With this purpose in mind, the argument structure presented in this chapter is organized into four levels of abstraction (illustrated in Figure 4.1):

Level 1 - Fit for Use: The top-level argument structure organizing what assets should be protected and how. In addition, the fit-for-use argument specifies pragmatic and other stakeholder-defined constraints and restrictions.

Level 2 - Attack Classes: An argument structure based on a taxonomy of classes of vulnerabilities, referred to as *attack classes*. Each attack class defines a general vulnerability that malicious adversaries can exploit to achieve a successful attack. A taxonomy of attack classes is specified for every asset identified in Level 1.

Level 3 - Decomposed Attack Classes: A structure for further attack class decomposition and refinement. Decomposed attack classes are necessary to bridge the gap between general security requirements and limitations of selected mitigation techniques.

Level 4 - Attack Class Mitigation: Arguments demonstrating how the software is mitigated against each defined attack class. In this context, methods for mitigating attacks are software modifications.

Because the top-level argument structure presents a fit-for-use claim, the entire argument structure is referred to as a fit-for-use (or fitness) argument. The triangular structure in Figure 4.1 illustrates how each subsequent level of argument refines, and consequently expands the argument. Within Level 1, the argument is relatively compact and simple. Proceeding through each subsequent level of argument adds more detail.

Generally, the four levels of the argument involve defining:

1. assets to protect (i.e., to secure),
2. threats against defined assets, and
3. the methods for mitigating defined threats.

While the use of assurance cases for security is at present minimally explored [33], argument structures with similar organizations have been discussed. Weinstock, Lipson and Goodenough [34,

35] provide an abstract argument structure for security cases and some general advice for building security cases where a similar decomposition of the argument is proposed. Courtney et al. [36] illustrate the application of a security case where the argument decomposition is based on attack classes. Security threats have a similar relationship to safety hazards. Consequently, there are similar argument patterns found within safety cases, discussed in the SafeSec standard [37, 38]

While these existing approaches provide some basis for initial investigation, they are used typically for system development and do not address issues specific for software modifications. The rationale of the argument structure presented in this chapter is based on:

1. elaborating and extending existing assurance argument structures for security,
2. adopting assurance argument structures from the safety community, and
3. presenting new argument structures where we are unable to find useful guidance.

The structure described in this chapter is not meant to be a definitive standard, but instead serves as a basis for discussion. The aim is (1) to increase the corpus of work on the topic of security cases, (2) to promote further discussion and debate as to how to effectively argue security, and fundamentally, (3) to present an argument structure with sufficient fidelity to support SSM. The remainder of this chapter describes the four levels of the argument structure individually in more detail.

4.2 Level 1 - Fit For Use

Security is not an isolated property. Consider a computer that is never turned on. Such a system would be highly secure, but fail to meet necessary functional correctness to be useful. Similarly, consider a software modification that requires gigabytes of additional memory or significantly alters original functionality. Even if the modification establishes desired security properties, stakeholders are unlikely to consider the software acceptable.

Instead of focusing just on security to determine acceptability of a software modification, stakeholders must also consider all competing constraints. In essence, the modified software must

provide adequate[1] security within pragmatic and other stakeholder-defined constraints. This concept of acceptability as is referred to as *fit for use* [23]:

Definition 4.1. ***Fit For Use***: *A system is fit for use if and only if it acceptably addresses a balance of stakeholder concerns.*

The fit-for-use concept is an adaptation of the "*appeal to requirements pattern*" introduced by Graydon [23] and is used as the top-level argument goal of the argument structure. The goal structure of the fit-for-use argument structure (i.e., Level 1) is illustrated in Figure 4.2. The remainder of this section further describes the rationale and components of this structure in more detail.

4.2.1 Argument Over Requirements

The top-level Goal (Goal 1.1) provides the fit-for-use claim. The remainder of the argument heavily depends upon the characteristics of the software that will be modified and in what environment the software will execute (referred to as an operating context). Context 1.2 references any documentation describing the operating context.

The strategy for arguing the fit-for-use claim is to argue over all requirements (Strategy 2.1), which includes security requirements, and requirements for all other competing constraints. Stakeholders might have these requirements specified in a separate document, which is referenced in Context 2.2. The types of requirements argued over are:

Goal 3.1 Security Requirements: Security requirements correspond to the ability for the modified software to thwart attacks to identified assets of importance. Adequately mitigating attacks to all defined assets constitutes meetings the security requirements. How mitigation is argued is heavily dependent upon key assumptions about what malicious adversaries might or might not have access to. These assumptions are referred to as the *threat model*, which are referenced in Context 3.2. All sub-arguments from Goal 3.1 therefore present security claims within the context of the threat model.

[1]The concept of "adequate" is intentionally vague, since it is entirely dependent on the stakeholder as to what constitutes adequacy within any particular context.

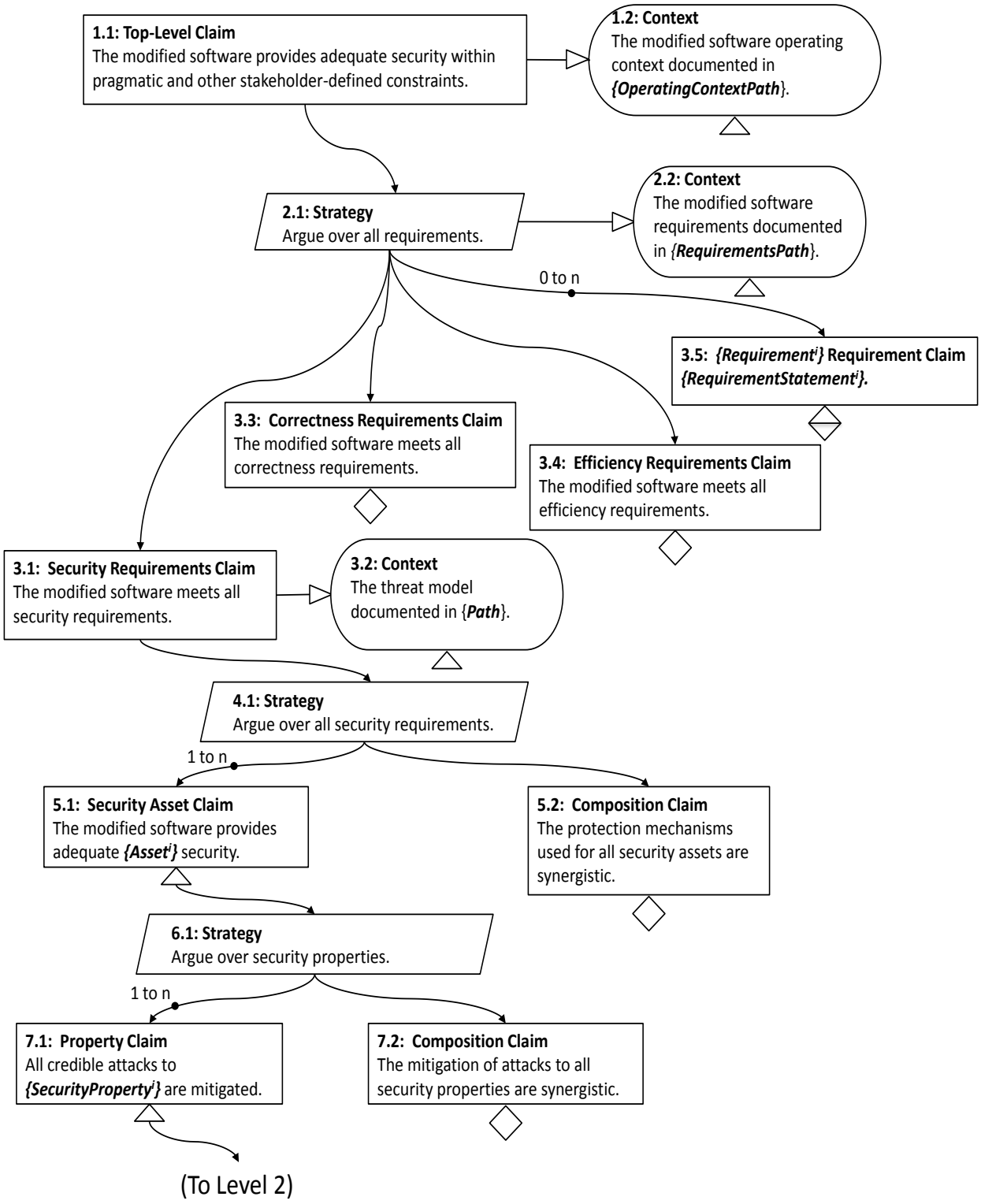


Figure 4.2: Argument level 1 — the Fit-For-Use argument structure

Goal 3.3 Correctness Requirements: Correctness requirements correspond to the ability of the modified software to operate as intended. Security requirements are often in competition with correctness requirements. For example, often to provide the most convincing security claims requires highly precise modifications, but high precision often negatively affects soundness. When a modification is precise but not sound, the functionality of the software might deviate too much to be considered fit for use by the stakeholders. Correctness requirements restrict the degree to which the intended operation of the modified software can deviate.

Goal 3.4 Efficiency Requirements: Efficiency requirements correspond to the ability of the modified software to operate within bounds of specified resource consumption overhead. Often security requirements are in direct competition with efficiency. In many cases, a highly effective security modification will increase consumption of memory or CPU cycles. Beyond a defined threshold, stakeholders might consider the software unfit for use. Efficiency requirements therefore restrict the resource consumption of the modified software.

Goal 3.5 Additional Requirements: Additional requirements correspond to any other requirements specified by the system stakeholders. Correctness and efficiency requirements are specifically identified in Figure 4.2 as they are the most common competing constraints; however, stakeholders can define any number of additional requirements. For example, stakeholders might have requirements corresponding to policies such as laws, regulations, internal/company policies, best practices, and standards.

The manner by which correctness, efficiency and other stakeholder-defined constraints are argued can vary greatly. Consequently, further structural decomposition of these goals is not specified. Instead, the remainder of the argument structure focuses on expanding on the security requirements claim (Goal 3.1).

4.2.2 Argument Over Assets

The need for security-enhancing software modifications stems from the threat that one or more software assets are at risk of being attacked. In Goal 5.1 of the fit-for-use argument structure (Figure 4.2), stakeholders define one or more security assets they wish to protect. A security asset is a software entity identified to have a security risk necessitating software modification. Assets can be data structures in a running program, or stored data on disk. Assets can also be services. The concept of a security asset is intentionally left generic to facilitate any interpretation. Example assets include:

- Credit card numbers
- Email messages
- Personal, financial, or medical records
- Web hosting or ftp services
- Data critical to system operation (e.g., in a medical device this could be data associated with dosage quantity and frequency)

The manner in which stakeholders determine which assets they want to protect can vary. In general, assets are chosen because there is a perceived threat of attack to the asset, and the asset has an inherent value. The value associated with each asset can vary, and consequently, how the assets are protected can also vary: i.e., the concept of "adequate" security is based partially on the value of the asset. Some example asset identification techniques include:

- Using a prescribed security requirement or standard, i.e., stakeholders might have previously documented the security-critical assets requiring protection.
- Responding to a perceived security threat to critical assets. For example, consider the recent series of attacks to US retailers Target, Neiman Marcus, and Michaels [6, 7, 8]. These attacks

targeted customer credit card numbers. Consequently, similar companies might perceive an increased threat to their customer's financial information.

- Analyzing common vulnerabilities. In this approach, users analyze vulnerabilities to determine what data is exploited by attackers. The information used to exploit the vulnerability becomes a security-critical asset.

- Traditional risk assessment, audits, and requirement elicitation techniques to expose security critical assets.

Each asset will eventually be protected using one or more mitigation techniques (software modifications). Because there can be variability in the types of techniques used to protect assets, there is a risk that chosen techniques will conflict/interfere. Goal 5.2 addresses this concern by arguing that the composition of protection mechanisms between assets does not interfere or result in any other undesirable effects, i.e., they are *synergistic*. While conceptually a single composition argument can be made for the entire collection of mitigation techniques, the remainder of the argument structure uses compositions arguments extensively to facilitate a more structured and rigorous approach to examining synergistic composition.

Definition 4.2. ***Synergistic Composition***: *Mitigation techniques have the quality of synergistic composition if all combined techniques do not interfere with each other or produce any emergent undesirable effects.*

4.2.3 Argument Over Security Properties

For each identified asset, stakeholders define what security properties must be established (Strategy 6.1). Security is traditionally described as a composite of three sub-properties: confidentiality, integrity, and availability (CIA), although other security properties might also be considered [39, 40].

Stakeholders are not restricted from defining their own security properties irrespective of existing concepts of security. The stakeholders are responsible for determining precisely what combination

of security properties is relevant for each asset. Security properties are specified in Goal 7.1. Mechanisms used to assure different security properties might conflict, hence, Goal 7.2 argues the composition of protection mechanisms between security properties is synergistic. The argument structure proceeds to Level 2 from Goal 7.1.

4.3 Level 2 - Attack Classes

Fundamentally, the objective of modifying software for security is to reduce and preferably eliminate successful attacks to important security properties of identified software assets. The argument structure leading up to and including Goal 7.1 in Figure 4.2 precisely identifies critical security assets and security properties. To demonstrate that attacks are adequately mitigated, however, requires an understanding of the various methods by which the asset(s) can be attacked [41]. The purpose of Level 2 of the argument structure is to further elaborate Goal 7.1 in Figure 4.2 by arguing that a set of *credible attack classes* are mitigated.

Definition 4.3. ***Attack Class***: *The set of all attacks that can exploit a specific software vulnerability.*

Definition 4.4. ***Credible Attack Class***: *A subset of an attack class consisting of all attacks for which there is a perceived and/or realistic threat of vulnerability exploitation.*

This level of argument can be thought of as arguing mitigation of threats or vulnerabilities; however, we explicitly choose the term "*attack class*" to focus the argument on the ultimate purpose of the modification, i.e., to mitigate attacks. The rationale for this strategy of argument is based partially on the security argument structure suggested by Courtney et al. [36] and the analogous structure of arguing over all credible safety hazards used extensively in safety cases [37].

This level of argument necessitates the development of a taxonomy of attack classes. A taxonomy of attack classes can be generated by the system stakeholders, for example by applying modified hazard analysis methods; however, generating a comprehensive taxonomy can be a

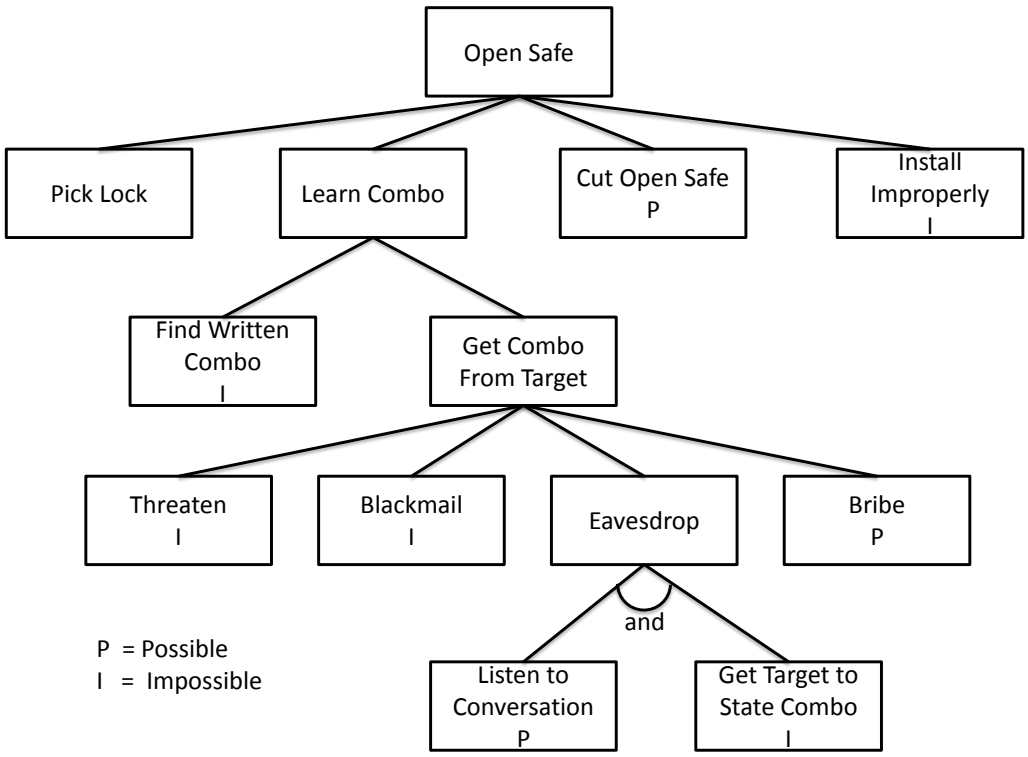


Figure 4.3: Example attack tree — figure taken from Schneier [41]

daunting challenge. Established taxonomies and attack patterns, such as Mitre's Common Weakness Enumeration (CWE) [42], the taxonomy of attack patterns described by Hoglund and McGraw [43], and the security flaws taxonomy of Landwehr et al. [44], can be used either to supplement an independently generated taxonomy, or as a complete and standalone taxonomy.

In principle, a chosen taxonomy can be organized into numerous sub-levels to reflect hierarchical relationships. Such relationships can describe an *attack tree* [45, 41]. An attack tree is a fault tree [46] for security, where the root of the tree documents a goal of an attacker, and the branches and leaves document different attack techniques that can be used to achieve the goal.

Figure 4.3, taken from Schneier [41], illustrates a basic example of an attack tree for attacks to open a safe. Nodes in the attack tree can be analyzed to determine how much of a danger the attack poses. In the given figure, a basic threat analysis is used to mark credible attacks with *P* (possible) and incredible attacks with *I* (impossible). Attack trees can also use AND nodes and OR nodes to indicate the relationship of attacks that must occur for the goal to be achieved.

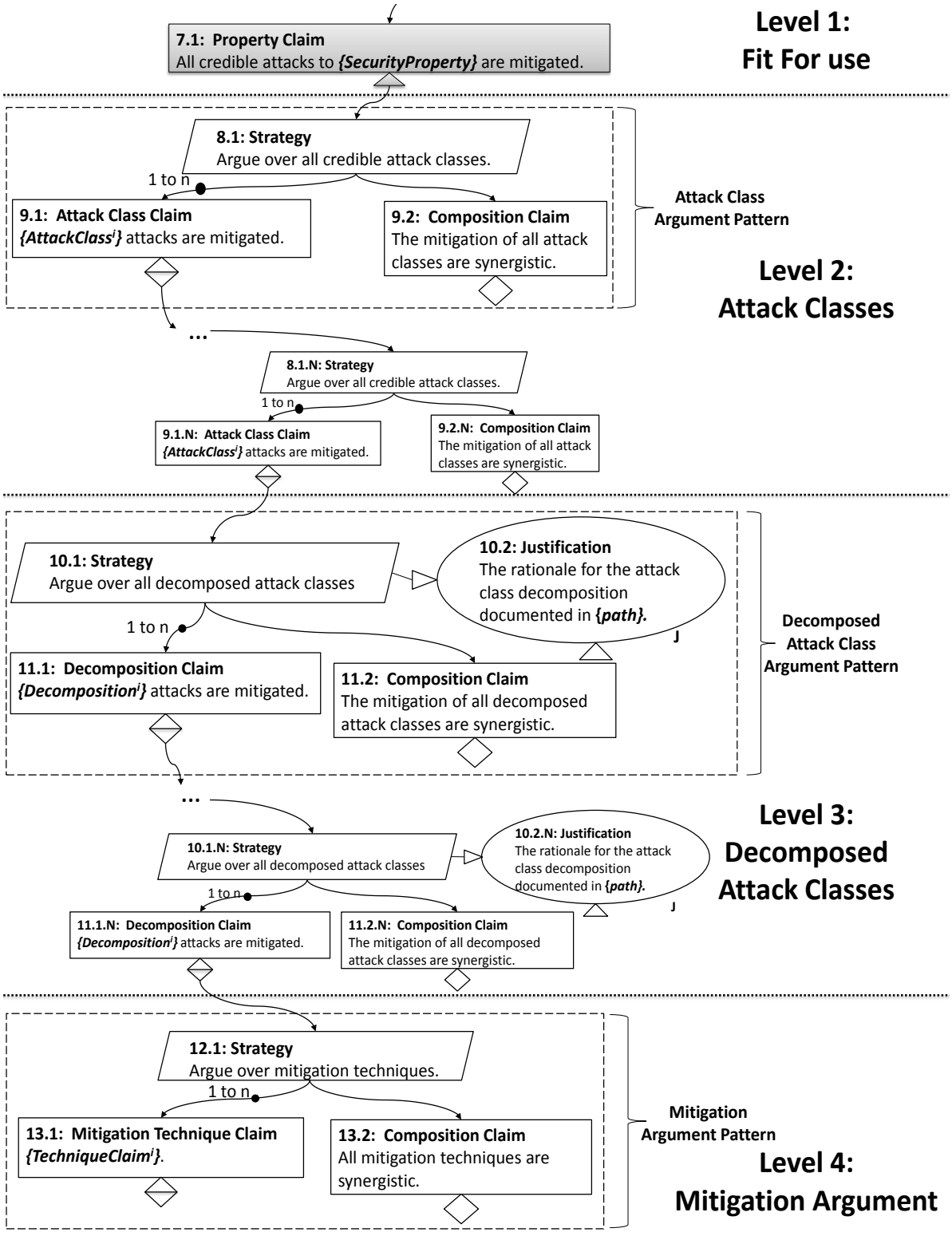


Figure 4.4: Argument structure for Level 2 (attack classes), Level 3 (decomposed attack classes), and Level 4 (mitigation argument)

The general argument structure used to document attack class taxonomies is shown in Level 2 of Figure 4.4. To express hierarchical attack class structures (attack tree structures), this structure can be composed with itself in a hierarchically recursive fashion (also illustrated in Level 2 of Figure 4.4). Goal 9.1 documents a mitigation claim for a given attack class.

4.4 Level 3 - Decomposed Attack Classes

Finding a mitigation technique that completely mitigates an entire attack class can be difficult. All mitigation techniques have benefits and weaknesses, not only in terms of the technique as a whole, but also with respect to trade-offs made during implementation and issues that occur as a result of applying techniques within the given operating context. In practice, a fundamental interplay exists between the details of a software security enhancement and the details of an attack.

As software modifications are examined by engineers, *weaknesses* or *limitations* (essentially "holes") in the defenses provided by chosen software modification techniques are often discovered. These limitations arise from a combination of characteristics of the subject system, the subject attack class, the subject asset and security properties, and the modification techniques to mitigate the attack class. These weaknesses often imply a further subdivision of the argument:

- one subdivision where the mitigation technique convincingly supports an attack mitigation claim, and
- the other where the technique fails to be convincing and will require additional support or an altered claim to be considered adequately mitigated.

In principle, a subdivision is necessary because the argument lacks sufficient specificity to justify the acceptability of a given mitigation technique.

As an example, consider a simple buffer overflow vulnerability. One way to protect data vulnerable to an overflow is to insert padding between the buffer and the vulnerable data. This protection works for naïve attacks in which the attacker is unaware of the padding. The protection

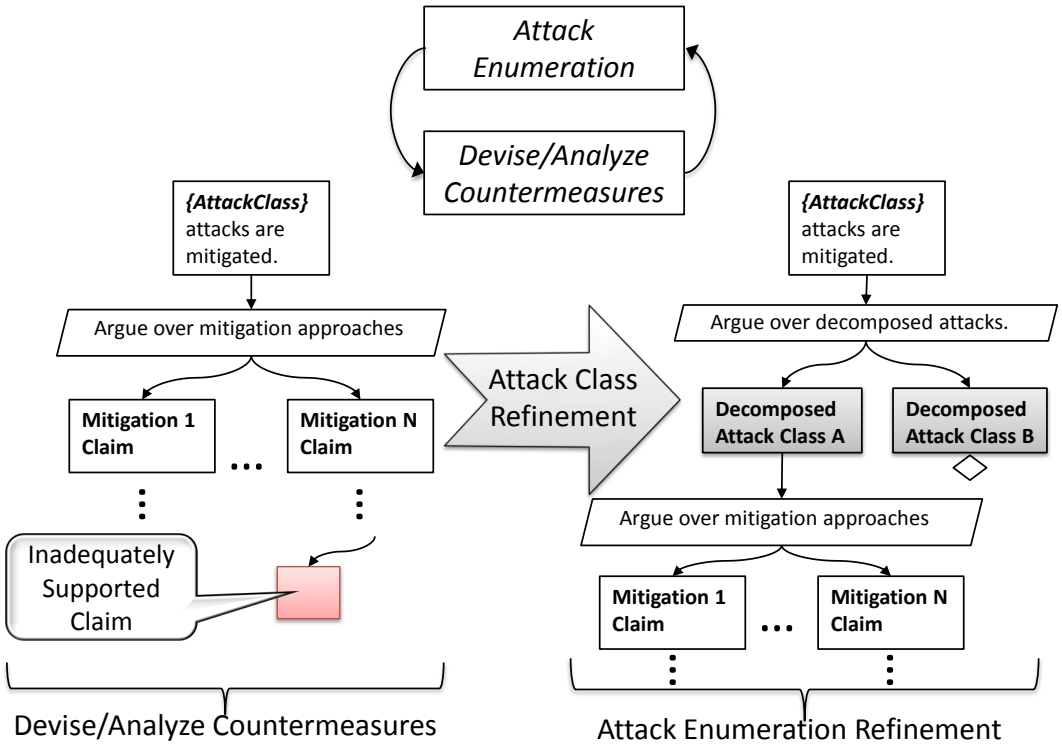


Figure 4.5: Construction process for decomposed attack class enumeration

does not work if the attacker floods the space following the buffer with a malicious value, because the flooding is bound to overwrite the vulnerable data eventually. This weakness of padding exposes a potential argument subdivision: one subdivision for attacks that flood memory and another for attacks that do not.

Argument Level 3 serves as a location for refinement of argument. Because level three is underneath the attack class level, Level 3 is said to "*decompose*" the above attack class enumeration. As weaknesses are exposed, and further subdivision of attack classes are identified, this level of argument is updated, i.e., the argument is repaired. Construction of the decomposed attack class enumeration is, in essence, performed by an iterative combination of top-down and bottom-up argument development. Analysis is performed on both the attack class that has to be mitigated and selected mitigation approaches. Analysis of attack classes drives the choice of mitigation methods, and analysis of mitigation methods potentially suggests new decomposed attack classes. These new decomposed attack classes must in turn be analyzed and mitigated. The iterative argument refinement cycle is shown in Figure 4.5.

While a decomposed attack class can refine an attack class from Level 2, a decomposed attack class is defined as any refinement to the argument as a result of limitations in chosen mitigation techniques. For example, a decomposed attack class can express a refinement of assets to defend.

Definition 4.5. ***Decomposed Attack Class***: *A refinement of the argument following the enumeration of attack classes in Level 2. The refinement is based on any limitations of chosen mitigation techniques.*

The primary benefit of Level 3 is that it makes explicit what selected mitigation approaches provide and, perhaps more importantly, do not provide. As new decomposed attack classes are argued using the structure shown in Level 3 of Figure 4.4, which closely resembles the attack class argument structure in Level 2. The key difference is the addition of a justification element (Justification 10.2) explaining why the subdivision was necessary. The justification highlights that the limitations of chosen mitigation techniques for easier review.

Instead of repairing the argument with decomposed attack classes, stakeholders can choose to discard a software modification technique if it does not completely mitigate a given attack class. Whether or not a given modification technique should be discarded or the attack tree refined is up to the stakeholders.

The key importance of decomposed attack classes is that engineers cannot be expected to know ahead of time that the given attack class taxonomy and general definition of assets will be sufficient. Further argument refinement might be necessary based on the characteristics of chosen mitigation techniques. Whichever modification approaches are finally chosen, Level 3 facilitates the clear and explicit documentation of the gap between stakeholder expectations and requirements, and the limitations of chosen mitigation techniques.

4.5 Level 4 - Mitigation Argument

Levels 2 and 3 primarily organize attack classes into hierarchical argument structures. Once the argument is sufficiently refined for a given attack class, an attack mitigation argument is constructed.

The final level of the argument structure, Level 4, provides the detailed mitigation argument for each attack class. In the context of this dissertation, mitigation techniques are analogous to software modifications. To argue an attack class is mitigated requires:

1. a precise determination of which configurations and combinations of mitigation techniques constitute adequate mitigation of the subject attack class, and

2. a clearly specified security claim for each selected mitigation technique.

The argument structure for arguing attack class mitigation is shown in Level 4 of Figure 4.4. For each "leaf" within the attack class hierarchy, Goal 13.1 is used make a mitigation claim. The precise wording for this claim cannot be specified generally. How an attack is mitigated can vary widely, and it is up to the stakeholders to define what claim is appropriate. In principle, one or more mitigation approaches can be used to mitigate any given attack class; hence, Goal 13.1 can be repeated under an attack class (once for each mitigation approach).

Determining which mitigation techniques should be applied necessitates a detailed analysis of which approaches are available and the benefits and weaknesses of each approach. As with deriving a taxonomy of attack classes, determining which mitigation techniques are appropriate can be a daunting challenge. While stakeholders can choose mitigation techniques in an ad hoc fashion until a convincing mitigation argument is made, another strategy is to organize mitigation needs into abstract categories. For example, mitigation methods can be classified as attack avoidance, removal, tolerance and forecasting [47]. These general categories could be further subdivided into more specific mitigation classes. Example detailed classes of mitigation include:

Increased attacker effort: Artificially diversifying a program can substantially increase the time required to achieve a successful attack. Evidence to support these mitigation claims would include the expected number of attempts required to achieve a successful attack together with the time required for each attempt.

Intrusion detection: Periodically checking for conditions indicating that an attack has occurred or is currently in progress can be used to effect a defense. Evidence to support these mitigation

claims would include the checking frequency, the probability of successful detection, and the probability of attacks occurring between detection.

Constrained asset access: Limiting or constraining how, where, and why an asset of interest is accessed can reduce the effectiveness of an attack. For example, data of interest can be made immutable or unreadable except by specific instructions. Evidence to support these mitigation claims would include probabilistic models of the expected reduction in the potential for attack.

Vulnerability removal: In some cases, the vulnerability leading to an attack can be removed by modifying the program. Evidence to support this mitigation claim would include program analysis showing that the vulnerability has been completely removed together with evidence showing that all attacks within the given attack class rely entirely on the fixed vulnerability.

While developing a general structure to organize attack mitigation techniques is outside the scope of this dissertation, Fægr and Hallsteinsen [48] provide a detailed taxonomy of security mitigation methods shown in Figure 4.6. This taxonomy can provide the basis for initial guidance and further research. In their taxonomy, mitigation methods are referred to as *solutions* which are organized into *tactics* and *patterns*. Tactics represent abstract mitigation methods while patterns are concrete and prescriptive security solutions. Tactics can hierarchically specialize other tactics, and the same specialization relationship exists for patterns. The specialization relationship is represented by a white triangle. Because patterns represent concrete solutions, patterns have associated documentation describing the problem, the solution and the impact the solution has on the system.

If the Fægr and Hallsteinsen taxonomy were expanded to include SOUP modifications, stakeholders would still have to determine:

- what combination of mitigation methods is appropriate,
- what mitigation claims are appropriate for each mitigation method, and
- find approaches to meet defined goals.

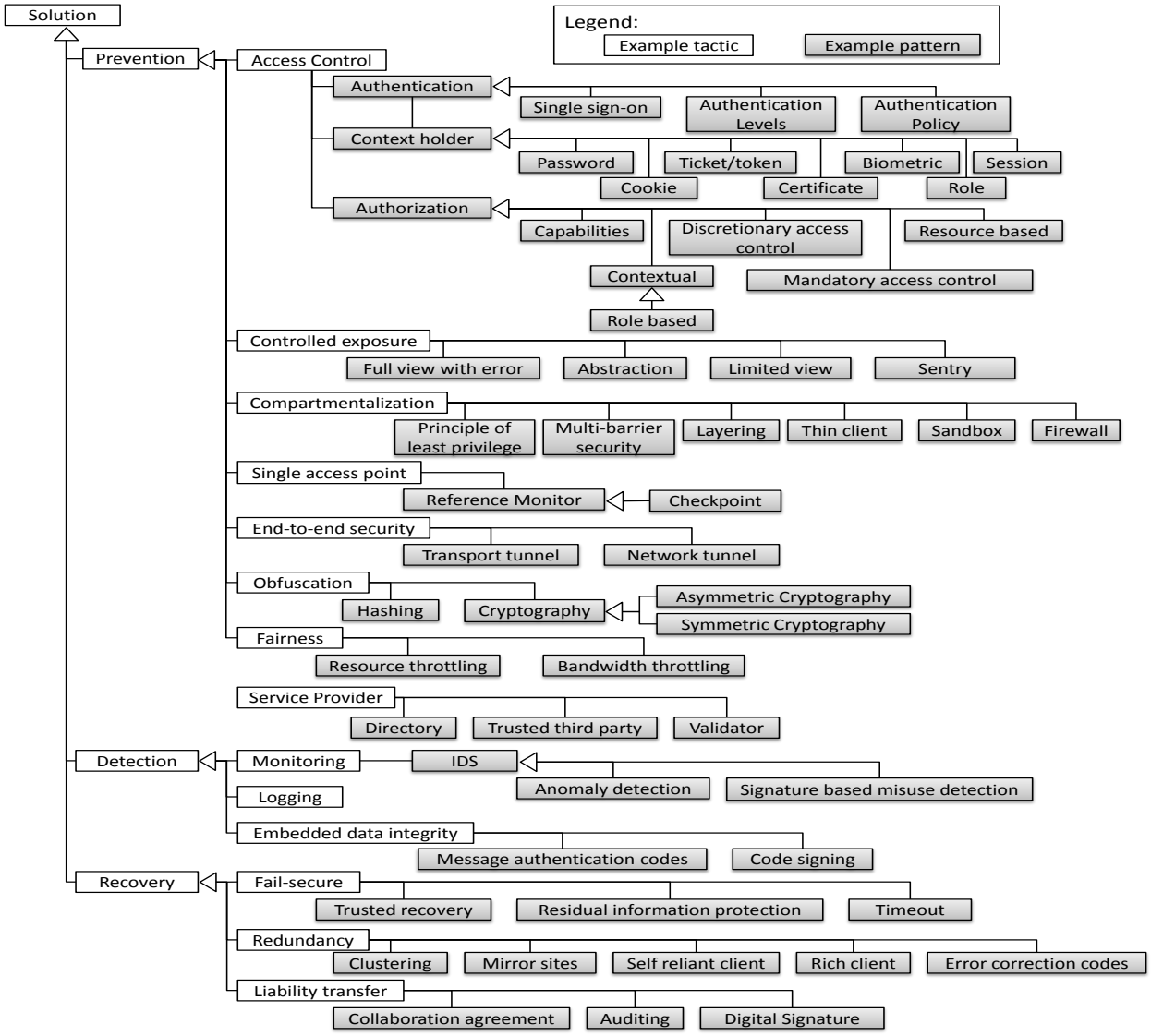


Figure 4.6: Example attack mitigation taxonomy – figure taken from Fægr and Hallsteinsen [48].

The key benefit is that a comprehensive and organized taxonomy allows stakeholders to elicit and examine their mitigation requirements. Further, by encoding the impacts of mitigation methods in the taxonomy, stakeholders can also determine what other areas of the argument would need to be reviewed if the approach is selected. For example, if a given approach is known to impact run-time overhead, arguments corresponding to the run-time efficiency will need to be reviewed.

Regardless of how stakeholders define mitigation techniques, each chosen mitigation technique must adhere to the threat model specified in Context 3.2 in Figure 4.2. Decisions as to which claim

to make for each mitigation technique must be made in consideration to the value of and perceived threat to the asset. Additionally, chosen mitigation techniques must adhere to constraints and other requirements defined in Goals 3.3, 3.4, and 3.5 in Figure 4.2.

Because of the wide variability in how an attack class can be shown to be mitigated, this level of argument is the limit of what can be specified in a general argument structure. Further subdivision of the argument structure is entirely dependent upon the mitigation methods used and the mitigation claim.

4.6 Considerations

Structure Flexibility

The fitness argument structure presented in this chapter provides a basic template and set of general guidelines for building arguments for security modifications. Stakeholders are free to alter the structure as necessary to meet their specific needs. For example, modified software might be used within a larger system and consequently, the assurance argument must fit into a larger assurance case, requiring some alteration to top-level claims and structures.

Argument Confidence

There are some additional considerations not specified in the given argument structure. The most apparent is the concept of *confidence*. Confidence refers to the degree of belief in the top-level argument claim; however, belief in a top-level claim is a summarization of belief in all sub-goals, which in turn is a belief in all contexts, argument strategies, and evidence throughout the argument. Example confidence concerns within the argument include:

- Were all appropriate software assets, security properties, attack classes, and mitigation techniques considered?

- Are all mitigation techniques appropriate, i.e., do they all abide by the assumptions within the threat model?
- Is the threat model appropriate and sufficient?
- Were all mitigation techniques implemented correctly, i.e., can all evidence in support of mitigation claims be trusted?
- Does a given item of evidence provide the necessary support for the given claim?

These issues can be summarized as concerns about the appropriateness, trustworthiness, and sufficiency of the argument. A further discussion of how to address confidence within the argument is given in Chapter 5.

Introduction of Vulnerabilities

Another concern not addressed explicitly in the argument structure is the potential for a software modification to add new vulnerabilities. Adding vulnerabilities can occur in one of two ways:

Faulty Implementation: The implementation of a mitigation technique contains exploitable vulnerabilities. For example, if a mitigation technique requires new buffers within memory but the buffer bounds are not checked, there is a potential for buffer overflow within the mitigation technique itself.

Emergent Vulnerabilities: The properties of one or more mitigation techniques can be exploited to achieve meaningful attacks. A simplistic example would be if a mitigation technique detects attacks and terminates the program upon detection. Attackers could use the mitigation technique for denial of service attacks.

Faulty implementation is a trustworthiness concern about the implementation which can be addressed using the confidence argument techniques discussed in Chapter 5. Emergent vulnerabilities are far more difficult to reason about and to prevent, except for trivial cases like the example given

above. The issue becomes more complicated when considering the possibility for emergent vulnerabilities resulting from the interactions between various mitigation techniques. We cannot provide an example of this behavior, neither can we demonstrate that such interactions are impossible.

Because emergent vulnerabilities are not well understood, no explicit goal arguing the absence of emergent vulnerabilities is given within the presented argument structure. To provide initial guidance, stakeholders could specify their own goal about the absence of emergent vulnerabilities using Goal 3.5 in Figure 4.2, or argue the absence of emergent vulnerabilities under the synergistic composition goals used throughout the argument.

For the purposes of this dissertation, expert opinion can serve as sufficient evidence about the non-existence of emergent vulnerabilities. Further investigation into mitigation interactions and side effects is left for future work.

Chapter 5

Assurance Assessment

In practice, assurance arguments include confidence arguments, e.g., arguments about sufficiency, appropriateness and trustworthiness [32, 49, 50, 51]. Argument structures that are presented throughout this dissertation (see for example Chapter 4) intentionally do not address confidence. Instead, we make the observation that confidence is a systemic and repeated concern throughout any assurance argument. To simplify and compartmentalize the argument, confidence arguments are extracted into separate arguments structures. In particular, we adopt and extend the concept of confidence arguments originally proposed by Hawkins et al. [32].

Fundamentally, the purpose of confidence arguments is to justify that an argument is valid (i.e., adequately supports a top-level claim). A top-level argument claim is statement about the general quality of the software system. Consequently, an assessment of argument confidence is both an assessment of the validity of an argument and also an assessment of the quality a given software system is claimed to have. As such, a discussion of how to document and address argument confidence also begs a discussion of how confidence can be used to assess software quality.

This chapter presents the rationale and structure of confidence arguments and a novel software quality metric framework based on assessment of confidence arguments [52]. Since security is the dependability property used for illustration in this dissertation, the metric framework is illustrated using security-critical systems specifically, i.e., the framework provides a metric of security. The

presented security metric framework, referred to as the Measurement Based on Security Argument (MBSA) framework, is a generic concept, exceeding the scope of this dissertation; however, MBSA motivates the rationale and form of confidence arguments, which are referred to throughout this dissertation and provides a direction for future research.

5.1 Overview

The goal of a security metric is to enable stakeholders of software systems to answer questions such as:

- How secure is my system?
- Is the software adequately secure for its use?
- How has a series of modifications affected my system's security?

Security metrics can facilitate enterprise-level design and operational decision making about software, but are difficult to capture [53, 54, 55, 56]. Without effective security metrics, software deployment decisions, upgrade decisions and so on cannot be addressed in an informed way for security-critical systems.

Measuring software security (i.e., the degree to which software copes with relevant security issues) presents a unique challenge because malicious action is difficult to model. Motivated adversaries actively seek to cause a failure, and typically attackers only need to find one vulnerability to be successful. Thus, an effective security metric must synthesize systemic data; a difficult task since software security is a complex and multi-dimensional property that includes interdependent sub-properties such as correctness, efficacy, and efficiency [55, 57, 58]. Further, security goals tend to differ from one organization to another and even among software systems within the same organization depending on the software's functionality and use [55].

Many methods for measuring software security have been proposed (for example, Herrmann [59] has consolidated over 900); however, stakeholders face a formidable challenge in applying use-

ful/appropriate metrics that capture their requirements. In dealing with this challenge, stakeholders must answer four questions:

1. which facets of the software system require measurement,
2. what kinds of measurements are appropriate,
3. how extensive should chosen metrics be applied, and
4. how should individual measurements be combined to provide a meaningful overall measure of security?

Of these four questions, the fourth is the most difficult to answer and most crucial, because there has to be a link between chosen metrics and stakeholder security requirements. In other words, for a metric to provide useful information to stakeholders, measurements must be linked explicitly to stakeholder security goals [56, 55]. Few existing security metrics are based on an explicit theory for this linkage.

We introduce the MBSA, a security metric framework in which security is measured in terms of degree of *belief* (i.e., confidence) in a security claim. The approach is to define a claim about the practical security of a system based on stakeholder requirements, such as freedom from one or more defined classes of vulnerabilities, and estimate the degree to which the claim can be believed. Typically, security-critical systems will be built with appropriate technologies, and various analyses and audits will have been undertaken on the system. Belief in the security claim depends upon the degree to which the development technologies, the analyses and the audits support that belief.

The key contribution of MBSA is that, to our knowledge, MBSA is the first framework for measuring security through measuring confidence (i.e., belief) in a security claim, based on analysis of the security argument for that system.

The remainder of this chapter further motivates the insight/rationale of the MBSA approach to security metrics (Section 5.2), and provides an overview of the problems with confidence in security arguments (Section 5.3). The mechanics of MSBA are described in Section 5.4, and an illustrative example of MBSA is presented in Section 5.5.

5.2 Metric Approach

Principle

The underlying principle of the MBSA approach to measuring security is to state a security claim about a system, develop an argument about why that claim should be believed, and then construct a metric based on the argument designed to assess *confidence* in the argument. More specifically, the metric is based on the following general principles:

Claim: The security *requirement* of the system's stakeholders is stated precisely as a security claim about the system. This claim is stated in terms of a particular context that defines the system itself, the system's operating environment, and the threat model to which the system is subjected. The claim is established carefully to document the stakeholders' fundamental security requirement. By beginning with a security requirement, MBSA does not need to measure security adequacy: adequacy is the basis of the security claim.

Evidence: The system is engineered to meet the stated requirement. The technology employed in the engineering and assessment of the system generates a body of *evidence* about the systems security properties. This evidence includes information about system tests, system analyses, vulnerability avoidance techniques employed, and so on.

Argument: In principle, the engineering of the system should ensure that the system meets the stated security claim. Doubts arise, however, in terms of the adequacy and completeness of the engineering and assessment, and so belief in the claim might not be warranted. Thus, in practice the critical issue facing the stakeholders is whether they can believe that the security requirement is met. Construction of a rigorous *argument* documents the rationale for belief in the claim based on the available evidence.

Metric: In practice, residual doubts in an argument are inevitable (see Chapter 2). For example, the argument might not have taken into account all possible circumstances or some elements of the argument might be based on invalid or corrupted data. MBSA computes a metric based

on the argument that reflects confidence in belief in the security claim, i.e., confidence in the argument. The metric is therefore an assessment of security: if the claim is true, the system is adequately secure. Any doubt about the claim is a reflection of possible insecurity, and stakeholders can make decisions about system deployment and operation based upon the degree of disbelief that they are prepared to accept. Stakeholders can also direct further analysis of the system based on exposed areas of doubt.

Metric Form

The metric computed to assess confidence in the security claim has to meet two goals:

1. The metric must take account of all of the factors that could influence belief in the security claim, i.e., the metric must be *complete*. Since the metric is derived from an argument, the metric would be incomplete if either the argument itself was incomplete or the argument, though complete, was not fully considered.

2. The metric must present all of the factors that could influence belief in the security claim in a way that enables a properly informed judgment of belief in the security claim, i.e., the metric must be *valid*. This goal requires that the metric present all aspects of possible doubt in the claim in a form that can be examined, judged, and acted upon by the stakeholders.

The MBSA framework addresses these goal using a two-step process:

1. The system security argument is annotated with confidence information about the various elements of the argument.

2. A function is computed with the annotations as input. The function produces a presentation of information that constitutes the metric.

5.3 Argument Confidence

In general, belief in the top-level goal (claim) of an argument, whether for safety, security or another property is important. The premises upon which the MBSA framework is based are:

1. The top-level goal in the security argument defines the stakeholders' security requirement adequately.

2. The belief in the truth of the top-level goal is justified by the argument.

Thus, security depends on the adequacy of the argument, and so meaningful security metrics are measurements of confidence in the argument. The MBSA framework is designed to measure argument confidence in a practical way, and present the results of the measurement to stakeholders in a manner that allows them to make deployment decisions about security-critical software systems.

Although belief in the top-level goal rests on confidence in the associated argument, confidence in an argument is an elusive concept. Dictionaries do not give a definition that can be used in an engineering context, deferring instead to the use of synonyms such as "trust" or "faith".

The U.K. Ministry of Defence definition of a safety case (see Chapter 2) requires a *compelling* argument, yet no definition of "compelling" is given. Graydon et al. have proposed a suitable practical definition [25]. Their definition is termed *operational* since they equate a "compelling" argument to an argument having a set of established and measurable properties.

Hawkins et al. have introduced the notion of *confidence arguments* to supplement safety and security arguments in order to capture confidence assessment [32]. A confidence argument supplements a traditional argument and supplies the rationale for belief in the quality of each of the argument's items of evidence, context definitions, and inferences. When these elements are added to an argument, there is an assertion that the element is valid and correct, i.e., the element serves the intended purpose to support a claim. Assertions in the argument are linked via an *Assurance Claim Point* (ACP) to the relevant confidence argument where confidence in the assertion is argued.

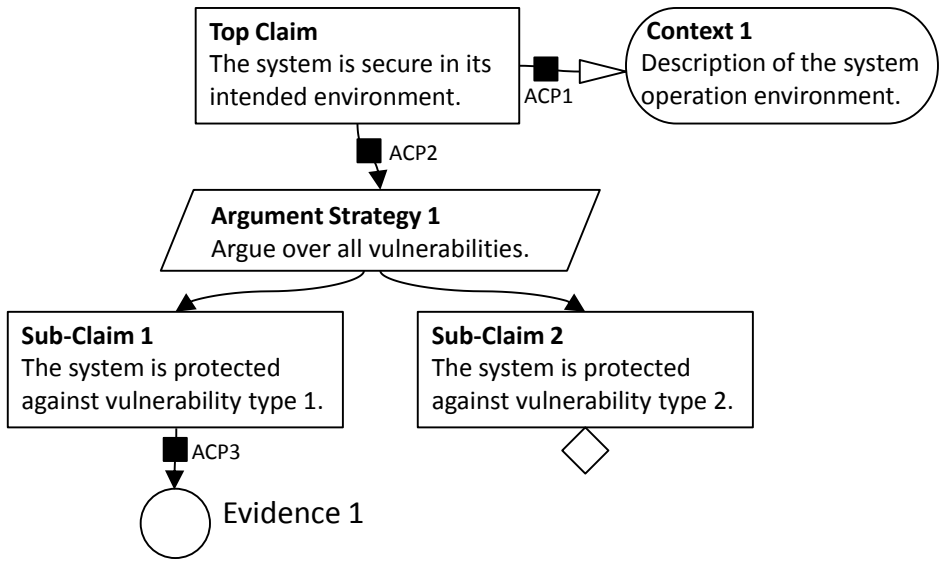


Figure 5.1: Example argument fragment in Goal Structuring Notation (GSN) annotated with with Assurance Claim Points (ACP)

The GSN extension to document ACPs is a black square, illustrated in Figure 5.1. For each ACP there is a corresponding confidence argument. The three points shown are associated with a context item (ACP1), an inference (ACP2), and an evidence item (ACP3).

In order to avoid the difficulties that arise with terms such as “compelling”, we have adopted the concept of an operational definition from Graydon et al. [25], i.e., a definition that defines confidence in a practical way based upon measurable or estimable argument characteristics. The operational definition begins with the adoption of the confidence arguments as defined by Hawkins et al. [32] and then elaborates this concept to provide an operational framework suitable for use in security metrics.

5.3.1 Sources of Doubt

Lack of confidence in arguments and associated doubt in the top-level goal arises primarily from inevitable doubt about the truth of the goals within the argument. Any goal within an argument might or might not be true, and belief in a goal is usually a matter of judgment. A goal thought to be true might not be true because:

- The inference upon which a goal depends is invalid, i.e., the argument strategy for justifying a goal might be inappropriate or incomplete.
- The evidence upon which the goal depends is invalid or inaccurate.
- The goal might be invalid for the defined context or the context definition might be inaccurate for the system of interest

When an argument inference is used to connect one claim to another claim or an item of evidence or a context is added to an argument to support or contextualize a claim, there is an assertion about the validity and accuracy of that element of argument. These assertions are in essence the sources of argument doubt.

As an example of how doubt can arise in an argument, consider a goal within a security argument which states that a given application is adequately protects against buffer overflow attacks. One form of evidence upon which the truth of this goal might rest could be the results of testing the application whereby the program is probed for buffer overflows. To support the truth of the goal with the results of testing is an assertion that:

1. the test results were correctly documented,
2. the test suite had sufficient coverage,
3. the tests were performed correctly,
4. the tests actually demonstrate an ability to thwart buffer overflow attacks,
5. the test conditions are within the intended operating conditions,
6. the test conditions represent realistic operating conditions, and so on.

5.3.2 Operational Definition of Confidence

In order to develop the operational definition of confidence that we require, we define confidence with respect to three properties:

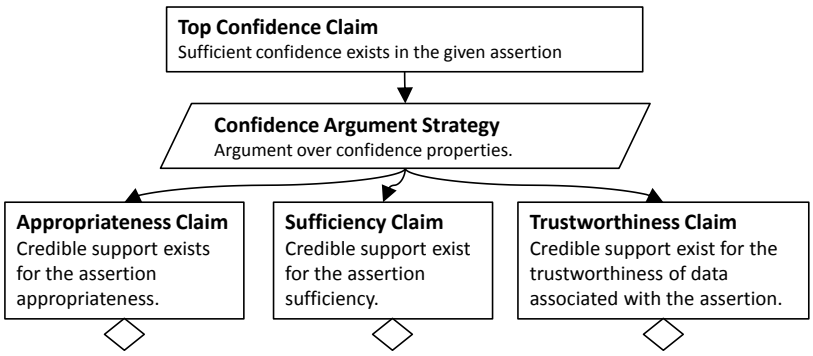


Figure 5.2: Confidence argument structure

Appropriateness Does the argument element perform the intended purpose (i.e., goal support or contextualization) and is it realistic?

Sufficiency Does the argument element achieve its purpose sufficiently, i.e., have all considerations that might affect the efficacy of the argument element negatively been considered and have all relevant concerns been handled?

Trustworthiness If the argument element relies on data of any kind, can the integrity or accuracy of the data be trusted, i.e., believed?

In our approach, we adopt the use of confidence arguments, based on the concept introduced by Hawkins et al. [32]. A confidence argument is constructed for each ACP in the security argument, i.e., for each assertion associated with contexts, items of evidence, and inferences. Each confidence argument demonstrates sufficient confidence in a given assertion by arguing the existence of our defined properties of confidence, illustrated in Figure 5.2. The strategy for how to argue each property is a topic of current research, and not defined here.

Confidence arguments are used by MBSA to generate confidence metrics for any claim or set of claims in the argument, discussed in the following section.

5.4 MBSA

The Measurement Based on Security Argument (MBSA) framework is in two parts that operate sequentially:

Confidence Assessment: Given a security argument where all assertions have corresponding confidence arguments, each confidence argument is annotated with a vector of measurements. Each measurement vector, referred to as a *Confidence of Assertion Vector* (CAV), contains a set of measures, the collection of which is meant to capture the quality/strength of the corresponding confidence argument. Each measurement in a CAV corresponds to a particular property of confidence and is derived from expert judgment in almost all circumstances.

Interpretation: Confidence metrics are presented to decision makers by interpreting the results of confidence assessment (i.e., CAVs). Given a security argument annotated with CAV data, interpretation contextualizes, simplifies, and presents CAV data at the request of decision makers to best facilitate decision making. Interpretation of CAVs is performed by a sequence of *interpretation functions*.

The remainder of this section describes the underlying form and mechanics of confidence assessment and interpretation.

5.4.1 Confidence Assessment

During confidence assessment, each confidence argument is evaluated by one or more technical experts, illustrated in Figure 5.3. The figure depicts a security argument where each confidence argument (represented by black squares) is sent to technical experts to evaluate based on their informed judgment. Confidence arguments are used to support belief in an assertion. Consequently, experts are chosen for their expertise in the type of assertion associated with the confidence argument as well as their expertise in the component of the software affected by the assertion.

Experts derive a vector of measurements to quantify the quality/strength of the confidence argument, referred to as a Confidence of Assertion Vector (CAV). CAVs are the basic unit within

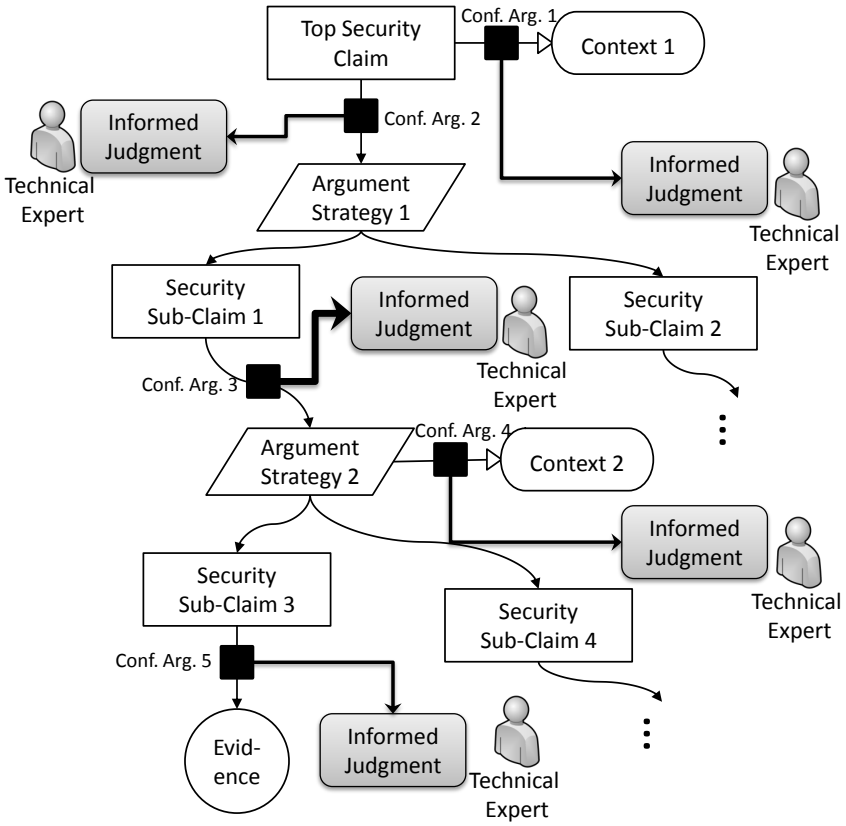


Figure 5.3: Confidence assessment

the MBSA framework from which all metrics presented to decision makers are derived. At the completion of confidence assessment, each confidence argument is annotated with a CAV.

The general form of a CAV is represented as:

$$\vec{C}^{a\_type} =< \mu_1^{m\_type}, \mu_2^{m\_type}, ..., \mu_n^{m\_type} >$$

Each CAV, $\vec{C}$, has an associated assertion type, a_type, the primary assertion types being: inference, context or evidence (see Section 5.3). Similarly, measurements within the vector, μ, also have a corresponding type, m_type, indicating what property of confidence the measurement is meant to capture, e.g., sufficiency. All CAVs of the same a_type contain the same number of type of measurements. Each individual μ can be a single value or vector themselves, potentially containing further sub-vectors.

The general CAV form is intentionally left abstract and flexible to facilitate any custom implementation; however, we propose an implementation based on a direct mapping to the confidence argument. Our proposed confidence argument structure is to argue over three primary confidence properties, i.e., appropriateness, sufficiency, and trustworthiness (see Figure 5.2). A corresponding CAV structure for $\vec{C}^{context}$, $\vec{C}^{inference}$ and $\vec{C}^{evidence}$ would map measurements to confidence properties as follows:

$$< \mu^{appropriate}, \mu^{sufficient}, \mu^{trustworthy} >$$

While this implementation of the CAV concept provides a structure for what measurements should represent, the precise form of values each measurement should take is not specified. Fundamentally the value produced for each measurement should be based on further examination of the strategy of argument used to demonstrate the confidence property. For example, an approach to argue the sufficiency of an assertion is to argue that all assurance deficits have been considered, and that all relevant deficits have been adequately handled. This kind of argument lends itself to measures of confidence in terms of Baconian probability [49].

Generally, measurements can take any form, which can be, in addition to Baconian probability, Bayesian probability, expert judgment, checklists, or even some combination of these measurements. Since the precise strategy of argumentation in confidence arguments is a topic of current research and debate (see Section 5.3) the exact form is not well established. Consequently, the precise form of each measurement type should take is also not currently established.

Finally, any measure of confidence will have residual doubts about the measure itself (i.e., meta-confidence); however, measuring confidence in this manner can be used to reduce residual doubts to be *As Low As Reasonably Practicable* (ALARP) [15]. Section 5.4.3 provides a discussion for potentially addressing meta-confidence concerns using MBSA.

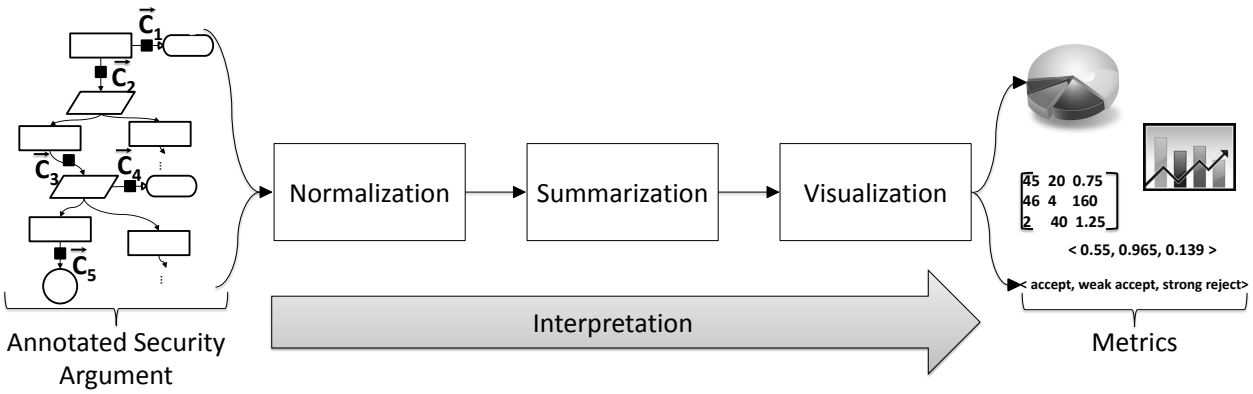


Figure 5.4: Interpretation of confidence assessment

5.4.2 Interpretation

When confidence assessment is completed, the security argument is annotated with CAVs for each confidence argument. Interpretation functions are then applied to CAVs to allow decision makers to derive various "views" (i.e., metrics) of the state of confidence in the security argument. Interpretation functions come in three forms that are applied in the following sequence (illustrated in Figure 5.4):

Normalization: Normalization functions standardize or contextualize each measurement stored within a given CAV. For example, security arguments frequently require amendments as concepts of security or the operating environment evolve over time. In response to these inevitable context changes, a normalization function can be developed to apply weights to (i.e., scale) affected CAVs to keep the results relevant within the new context. Other examples of normalization include standardizing the scale of measurements (e.g., between 0 and 1), filtering pertinent data, and contextualizing measurements with respect pass/fail thresholds.

Summarization: Summarization functions are used to distill CAVs into simpler and more easily comprehended forms. For example, to determine how much confidence there is in a given claim would require examination of all CAVs stemming from that claim and all its sub-claims. Instead of evaluating all CAVs individually, summarization can be applied to provide a simpler measurement of confidence for the given claim, such as an average, dot product, magnitude,

min, max, standard deviation, etc. Summarization functions can operate on individual CAVs, a collection of CAVs, or a collection of previously summarized data. The data resulting from summarization can take almost any form, including individual scalar values, a single CAV, or a collection/matrix of CAVs.

Visualization: Once CAVs have been normalized and summarized, one or more visualization functions can be applied to display the results to decision makers. Visualization can be used to produce graphs, charts, tables, annotated assurance cases, raw numbers, etc.

Generally, all CAVs are processed by these functions in the given order; however, decision makers might choose not to implement one or more of these interpretation functions. For example, decision makers could choose to view all raw CAV results without any interpretation, view raw summarizing numbers without visualization, view normalized CAVs without summarization, etc.

No single method for implementing and combining interpretation functions is defined. Decision makers are given the freedom to choose dynamically how to generate metrics. By altering interpretation functions, decision makers can alternate between very abstract and highly detailed measurements. Decision makers can also alter the parameters of interpretation functions to observe how hypothetical alteration to the software might affect overall security. As such, a practical implementation of MBSA would require users to define a database of interpretation functions. Once defined, decision makers can quickly switch between views.

5.4.3 Meta-Confidence

Even with a measure of confidence, there are always remaining doubts about the confidence one should have in that measurement, i.e., meta-confidence. Much like doubts in a given security argument, confidence measurement doubts arise out of assertions made in measuring confidence. An attempt to measure meta-confidence results in further assertions; hence, one could attempt to measure confidence about confidence ad infinitum.

A potential solution to infinite regress is to assign an additional team of experts to construct a meta-confidence vector which captures the residual doubts. The primary doubts about metrics produced by MBSA are:

1. doubts about the correctness of the framework implementation (e.g., the interpretation functions),
2. doubts about the qualifications of experts and
3. doubts the integrity of the measurement data.

Experts could therefore evaluate these doubts in the following meta-confidence vector:

$$\vec{C}^{meta} =< \mu^{implementation}, \mu^{expert}, \mu^{data} >$$

This concept of meta-confidence can serve as an initial operational definition of meta-confidence and can be supplemented as other doubts are discovered to meet the needs of those implementing the framework.

The concept of this approach is to provide decision makers with a value a metric where all doubts have been reduced to As Low As Reasonably Practicable (ALARP) [15]. Decision makers could doubt the meta-confidence vector, but to continue measurement beyond this point will require measuring the same general characteristics. Hence, while confidence metrics to this point have been beneficial in reducing the problem into a smaller and more manageable set of concerns, the concerns of meta-confidence cannot be further reduced and consequently the benefit of continued measurement is minimal or non-existent. We use this concept of irreducible or non-simplifiable doubt as an operational definition of ALARP as a potential solution meta-confidence concerns. The precise nature of the form of individual meta-confidence metrics is left for future work, and we omit discussion of meta-confidence from the remainder of this dissertation.

5.5 Illustrative Example: The Stoplight Metric

We illustrate the general mechanics and uses of the MBSA framework with an example security metric that we call the *stoplight metric*. In this scenario, a decision maker must decide if a given software system is adequately secure to ship, and if not, needs to make recommendations as to what components of the software system need to be improved. The decision maker has at his or her disposal a security argument for the software system that is complete with confidence arguments. For this example, the exact nature of the given software system is not important and is therefore not specified.

The decision maker requires a means to evaluate the security argument systematically using a series of measurements of confidence. By selecting alternative views of the state of the system, the decision maker can affirm their decision and also isolate areas of highest concern. As such, the decision maker first uses a very terse and cursory metric to get an idea of the general confidence of the entire argument, referred to as the stoplight metric.

The premise of the stoplight metric is that each goal (claim) in the security argument can be described in terms of three levels of confidence, corresponding to three colors:

Green: "Pass"; acceptable confidence

Yellow: "Pass Pending"; provisionally acceptable confidence

Red: "No Pass"; unacceptable confidence

While the stoplight metric can be used to provide a single color for any given claim, the decision maker desires an overview of the entire argument to understand the systemic security strengths and weaknesses.

5.5.1 Application of MBSA

To implement the stoplight metric using MBSA, first every confidence argument is assessed and assigned a CAV by teams of experts. For purposes of this illustrations, CAVs will have the

form proposed in Section 5.3, i.e., provide measurements of trustworthiness, appropriateness, and sufficiency. As mentioned in Section 5.4, the form of these measurements is a topic of further study, so for this example, each measurement is in the form of a number between 1 and 10, inclusive.

With CAVs for every confidence argument, we define a stoplight algebra for assigning confidence status (Green, Yellow, Red) to a claim through the use of interpretation functions. The sequence of interpretation functions is defined as follows:

Normalization: For each CAV, a threshold is applied to each individual measurement within a CAV to translate numeric measurements into one of the three stoplight colors. The choice of thresholds is entirely up to the decision maker. For this example, measurements greater than or equal to 7 are colored Green, measurements above 4 but below 7 are colored Yellow, and measurements that are less than or equal to 4 are colored Red. For example a CAV with values $< 8, 2, 5 >$ would be normalized into a new CAV with values $< Green, Red, Yellow >$.

Summarization: Summarization is accomplished through first summarizing each CAV into a single color (referred to as a CAV summary color) and then percolating CAV summary colors up through the argument to each claim (i.e., from leaf goals up to the top-level claim). Percolation is another type of summarization function in which a claim is assigned color based on summarizing all CAV summary colors from that claim and all its sub-claims. Both summarizing functions operate using the same color algebra:

- When all values being summarized are Green, the result is also Green.
- If the values being summarized contain a mixture of Yellow and Green, but no Red, the summarized value is colored Yellow.
- If any values being summarized are Red, regardless of the color of the other values, the summarized value is colored Red.

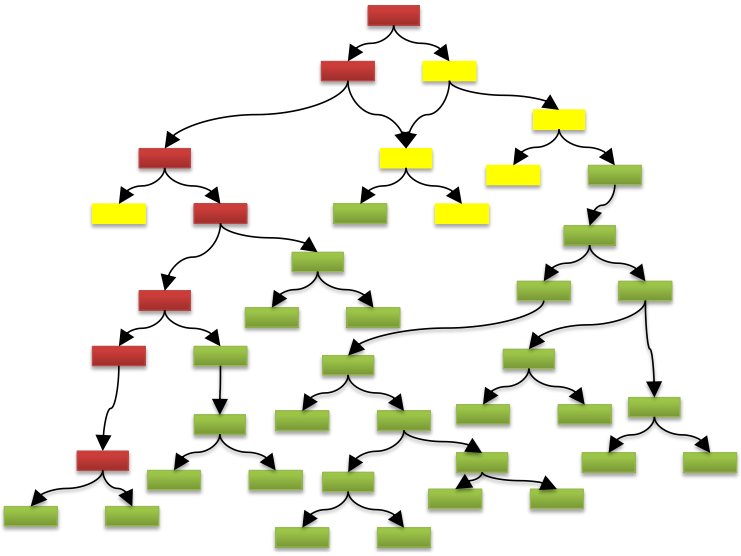


Figure 5.5: Stoplight metric example

For example, a CAV with normalized values $< Green, Red, Yellow >$ would be summarized as Red. The CAV summary color Red would then be used in the percolation summarization function with other summarized CAV colors.

Visualization: The visualization chosen for the stoplight metric is to present a miniaturized view of the security argument, where the argument has been stripped of all elements except for claims, and each claim has been colored. A claim is colored based on the summarized color value from that claim and all its sub-arguments. An example visualization is shown in Figure 5.5.

5.5.2 Uses of the Stoplight Metric

The stoplight metric provides a quick method to assess the entire state of security of a software system. For example, in Figure 5.5, the top-level claim is colored Red, hence, sufficient confidence in the primary security claim does not exist. The claims on the left side of the argument can be seen to be unacceptable and therefore requiring immediate attention.

The stoplight metric provides a initial view of the security of a software system, not a definitive measurement. For example, if the top-level claim were colored Green, a skeptical decision maker would be unwilling to accept that result without first posing some additional questions such as:

- What if a claim's color was changed from Green to Red, or vice versa?
- What if the color thresholds were altered?
- What if portions of the argument were considered higher priority than others?

MBSA facilitates these answering questions by allowing decision makers to switch between different metrics, or change the parameters of a single metric by using different interpretation functions: i.e., MBSA facilities performing a *"what if?"* analysis.

5.6 Scope of MBSA

The uses of MBSA far exceed the scope of SSM. In fact, the principles likely extend beyond the scope of measuring security to measuring any property for which an assurance case is provided. The potential power and use of the concepts of MBSA is compelling and raises many research questions such as:

- Are trustworthiness, appropriateness and sufficiency the correct properties to measure?
- Can measuring confidence be standardized?
- Does the approach facilitate making more secure systems (i.e., a longitudinal study is needed across many different systems, both using the metric and not using the metric)?
- Can experts be expected to provide meaningful and repeatable measurements?
- Can confidence be measured by means other than expert opinion?
- Can measuring confidence be used to address the issue of determining compositional correctness of arguments in an automated or semi-automated sense?

Not only does the scope of these questions lead to a field of research beyond the scope of this dissertation, this concept of measurement would require a shift in how confidence is addressed within systems already using assurance cases. The issue within security is that presently security

cases are minimally explored. The success of assurance case technologies in the safety community would indicate the applicability and feasibility of the use of assurance cases for security is likely. Many of the questions posed about MBSA cannot be answered with empirical data until more organizations adopt the use of security cases for their products, and make their data public.

Within the scope of SSM, there are many potential uses of MBSA, such as:

- A metric to find weaknesses in a security argument demonstrating the need for SSM principles (i.e., to determine the applicability of SSM).
- A means for assessing an assurance case instance (see Chapter 3). In this context, MBSA can serve as a means for assessment within the finalization phase of SSM.
- A means for assisting strategy selection during the SSM process. For example, instead of selecting strategies based on predefined approaches, the selection can be based on selecting strategies for which the corresponding argument has the highest confidence.

Addressing each one of these uses of MBSA also exceeds the scope of this dissertation. We leave further exploration into the MBSA concept for future research.

5.7 Related Work

Many security metrics have been proposed. For example, Hermann [59] discusses over 900 different security metrics. The MBSA framework is based on argument confidence, a completely different concept from those used previously. Metrics have inherent assumptions and abstractions that have led some to conclude the need for a *meta*-metric to measure these risks [54]. MBSA addressing this issue.

The key to providing meaningful metrics is the linkage of evidence to a security goal [56, 55]. MBSA utilizes security arguments to structure security goals for the purpose of a security metric. The key benefit of this approach is the flexibility that it provides stakeholders to choose what constitutes adequate security for their purposes.

Other approaches that link metrics to goals have been proposed in which metrics are defined that focus on a set of desired security-critical properties [56, 53]. These approaches have to consider multiple forms of measurement to justify that the software has a desired security property. Such metrics rely on implicit arguments about why a security property should be measured, and what measurements will show that the desired property is present. MBSA does not necessarily obviate the need for these metrics, but provides a framework in which justification for a metric is made explicit.

Denney et al. have proposed an approach to quantifying confidence in assurance arguments using Bayesian Belief Networks (BBN) [60]. In this approach, confidence is measured by identifying components of the argument and measuring the confidence in those components probabilistically. The probabilities are percolated through the argument using the mechanics of BBNs. A BBN confidence metric prescribes a single approach to modeling and measuring confidence. The approach relies on belief in the various probabilities used and might abstract too much information for decision makers. MBSA is a more general framework that does not prescribe methods for assessing and percolating confidence data. Hence, probablistic confidence measurements and the use of BBNs to interpret and percolate those metrics is one potential implementation of MBSA.

Chapter 6

Selection Argumentation

Traditionally, an assurance case documents the rationale for belief in the acceptability of a single software system. To support the SSM process, however, the assurance case must express a range of modification variability as well as the conditions restricting which variations are acceptable (i.e., the argument must document the acceptable solution space—Chapter 3—and how to navigate it). In essence, the SSM assurance case must serve as a "blueprint" of acceptability that specifies how a fit-for-use argument is constructed for every software modification generated by the instantiated SSM process. When the SSM process generates a modification, the assurance blueprint is used to govern how an acceptable modification is constructed.

This chapter introduces the concept *selection argumentation*, a set of argument notation enhancements and techniques used to express within the SSM assurance case a fitness argument blueprint. Selection argumentation is used to capture the mechanics of the SSM process within an argument structure. Specifically, selection argumentation expresses a selection process for constructing a fitness argument that is tailored to a specific instance of SOUP, referred to as an *assurance case instance*. This chapter describes the core concepts and mechanics of selection argumentation, guidelines for the application and interpretation of selection argumentation, and an illustrative example.

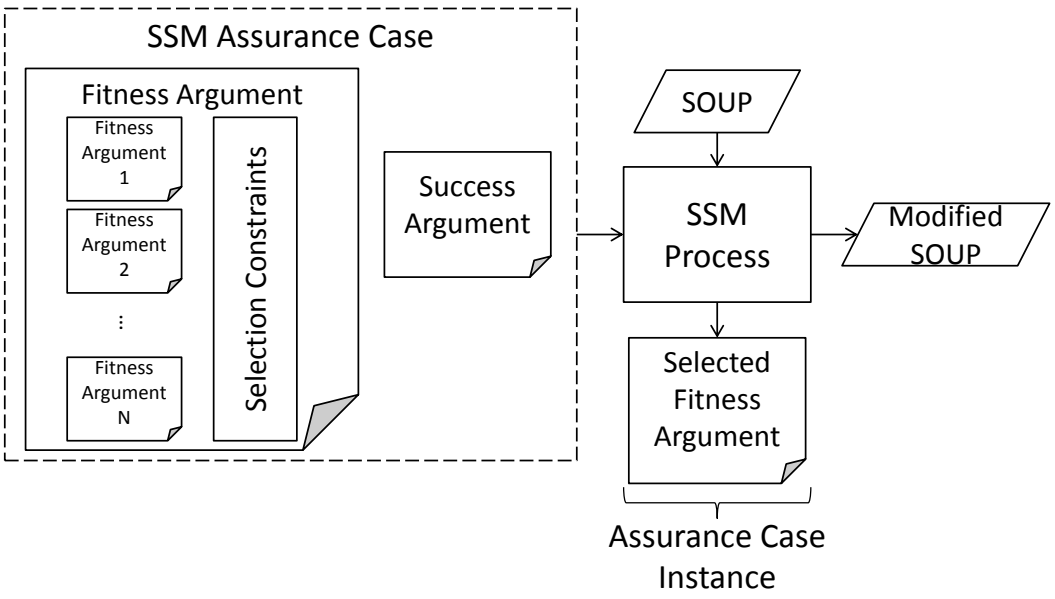


Figure 6.1: SSM assurance case mechanics

6.1 Overview

Because of the variation in how modifications are applied, the fitness argument within the SSM assurance case expresses a range of acceptable arguments and the constraints for selecting between argument alternatives. Modification variability is a result of the inability to produce an adequate argument that a modification will be fit for use. More specifically, some evidence necessary to justify a fitness argument alternative is not known prior to applying a modification. SSM argument developers must often assume (i.e., speculate) key evidence will be available and be of sufficient quality when a modification is generated. Assessments are performed within the SSM process to validate that the evidence exists and is of the desired quality. Assessments also determine if necessary evidence can be generated within developmental limits as specified within the success argument (see Chapter 3). How each fitness argument alternative is selected and assessed are described within the fitness argument in the form of selection constraints.

The SSM process is a realization of the fitness argument selection constraints. As a software modification is generated, an individualized fit-for-use argument is dynamically selected for the given modification, illustrated in Figure 6.1. The selected fitness argument is referred to as an

assurance case instance.

Definition 6.1. ***An Assurance Case Instance***: *An assurance case instance is a single assurance argument (i.e., a single fitness argument) selected from potentially many arguments. Selection is based on a set of predefined selection constraints and decision models.*

An assurance case instance can be supplied with any generated SOUP modification to justify the modification's acceptability; however, whether or not an assurance case instance is actually generated as an output by an instantiated SSM process is at the discretion of those implementing the model.

Of key importance is that SSM generated modifications are based on the selection of an assurance case instance.

Generally, the number of possible assurance case instances can be large, prohibiting a simple enumeration of all individual fitness arguments that are possible, as suggested in Figure 6.1. Instead, we propose the construction of a single argument structure that expresses variability in how argument elements and fragments are instantiated and combined.

Traditional assurance case technologies, however, do not sufficiently express the necessary argument variability and selection mechanics described within the SSM process. To address this challenge, this chapter introduces the concept of *selection argumentation*. Selection argumentation is used within a SSM fitness argument (see for example Chapter 4) to expresses the acceptable solution space and the criteria used by the SSM process to navigate the solution space.

6.1.1 General Mechanics

The mechanics of selection argumentation are based on the interactions between three fundamental concepts within the SSM model, illustrated in Figure 6.2:

The Acceptable Solution Space: The range of modification variations (i.e., SOUP modifications) that users are willing to accept or tolerate (i.e., the range of fit-for-use modifications).

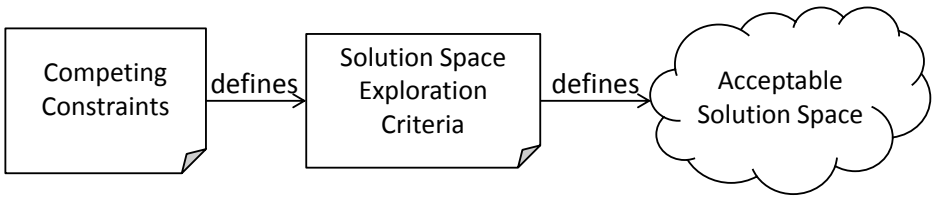


Figure 6.2: Fundamental SSM concepts

Solution Space Exploration Criteria: Decision models describing how the solution space is navigated. Because the exploration criteria govern how an acceptable modification is generated, the exploration criteria define the acceptable solution space.

Constraints: Evidence that must be generated and/or validated to justify a modification is fit for use. The decision models in the exploration criteria are defined based on the possibility of not generating necessary evidence (i.e., not meeting necessary constraints).

In context of a fitness argument, the solution space is a set of predefined argument structures for constructing an assurance case instance. The solution space is navigated based on the evidence that must be generated and/or validated within the fitness and success arguments. Selection argumentation expresses the above concepts within a fitness argument. As a result, the argument defines a selection process for constructing an assurance case instance.

Conceptually, selection argumentation describes the selection/development of an assurance case instance in terms of a tree traversal process. The intuition of this approach is that typically an argument documented in Goal Structuring Notation (GSN — see Chapter 2) is developed by a systematic and recursive traversal and development of the argument starting from the top-level claim down through all sub-claims. Traditionally, argument development is an iterative process, in which developers perform the following activities:

- choose one or more undeveloped claims,
- determine a set of alternatives to support these claims,
- assess alternatives, and

- proceed to construct the argument as preferred alternatives are deemed acceptable.

As a branch of argument is completed, developers traverse back up through the argument to the next undeveloped goal and perform the above activities again.

The development of an assurance case instance involves the same argument development activities, with one key exception:

Acceptable alternatives, how to choose between them, and constraints affecting the validity of each alternative are all predefined.

Generating an assurance case instance is therefore based on predefined rules. Selection argumentation expresses predefined development activities explicitly within the argument structure itself.

6.1.2 Components

Selection argumentation consists of three primary concepts and techniques used in the construction of a fit-for-use argument:

Product Line Argumentation: Variability in how an assurance case instance is constructed can be expressed by adopting and extending a proposed GSN argument notation for *software product lines* [22]. The proposed notation (referred to as product line argumentation) provides support for capturing the acceptable solution space within an argument.

Argument Guards: An argument guard expresses within a GSN argument the criteria that must be valid (i.e., the form of evidence that must be generated) in order for a sub-argument or other argument element to be included in an assurance case instance. Guards capture the constraints that affect how the solution space is explored (i.e., constraints restricting the selection of arguments).

Decision and Refinement Models: SSM involves the concept of making decisions about how to analyze and modify software, and then refining decisions if these decisions cannot be

fully supported (i.e., necessary evidence could not be generated). These decisions directly correspond to structures within the fit-for-use argument. To express decision and refinement mechanics within the argument requires that the argument encode decision and refinement models. Selection argumentation combines the use of product line argumentation with decision and refinement models to capture the solution space exploration criteria.

The remainder of this chapter describes the motivation, syntax, and semantics of each selection argumentation concept followed by general guidelines for the application and interpretation of selection argumentation within an argument. This chapter concludes with a brief illustrative example.

6.2 Product Line Argumentation

As methods for analyzing and modifying the software are selected in the SSM process, the result is a range of variability in the manner in which the software is modified. The modification variability is reflected in the variability in which claims are used to support a fit-for-use argument. The granularity of variability is classified into two general types:

External Variability Variation in the methods of analysis and modification and how extensively they are applied. This kind of variability is outside (external to) any given analysis or modification approach. For example, speculative analysis could have an option to choose static analysis or dynamic analysis, but not both. The use of one analysis technique rather than the other might depend on how well each analysis performs and therefore will vary depending on the nature of the software being analyzed.

Internal Variability Variation in how a single analysis or modification technique is configured and applied. This kind of variability is within (internal to) a single analysis or modification method. For example, a modification technique could involve checking data structures for corruption at run time to detect an attack. Frequent corruption checks might increase run-time

overhead to an unacceptable degree for some pieces of SOUP but not others; hence, the frequency of checks might vary, although the general approach is the same.

To allow an SSM assurance case to encode either kind of variation, selection argumentation adopts notation techniques applied to software product line engineering. This section provides a brief overview of the concept of product lines before describing product line assurance cases in more detail.

6.2.1 Product Line Overview

Often when a developer releases a software product, the product shares many characteristics (for example the architecture, core functionality, and risk mitigation methods) with other products offered by the same developer. Software product line engineering (SPLE), also called product family engineering, is a paradigm intended to improve productivity and decrease development time and costs in this scenario [61, 62, 63]. The premise of SPLE is to make maximum reuse of preexisting *core assets*, such as common architectures and software components. A software product line specifies not only what core assets can be used in development, but also the manner in which the assets can be composed. A software product generated by a product line is referred to as a *variation*.

Definition 6.2. ***Software Product Line Engineering (SPLE)***: *A software engineering paradigm for producing software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* [61].

Software product lines have also been proposed for developing software with variable dependability properties, specifically security [48]. In this context, a product variant is constructed by developing a software system using different property establishing mechanisms and configurations (e.g., varying attack mitigation techniques).

The principles of SSM share some characteristics in common with SPLE:

- SSM reuses the same SOUP analysis and modification techniques to effect a modification for any given SOUP. These analyses and modification approaches can be thought of as reusable core assets.

- SSM applies analyses and modification techniques in different ways depending on the characteristics of the SOUP being modified. Since each produced modification is different but shares some similar characteristics, modifications are analogous to the concept of variations in software product lines.

SSM and traditional SPLE differ in terms of their respective goals and uses:

- The ultimate goal of a SPLE is to improve development productivity [62]. While SSM principles might also improve productivity, the primary goal of SSM is to account for uncertainties in modifying software and produce software modifications within a range of tolerability.

- SPLE is typically used to generate new software products, where SSM is used to modify existing software.

- SPLE typically defines the mechanics to produce variations for different stakeholders and/or operating contexts. Conversely, an instantiation of SSM produces variations (modified software) for the same operating context and stakeholders.

- A product line captures and restricts the variability in software products; however, a variation is always meant to meet the needs of the stakeholders fully. In this sense, variations are intended to be equally acceptable for use. SSM differs in that modified software might not be equally acceptable, e.g., a given modification might provide better security guarantees than another. This can occur when assessments within SSM invalidate a preferred strategy, resulting in selecting a less preferred strategy. Hence, SSM not only captures and restricts variability in modifications and the dependencies associated with each variation, but SSM

also includes the dependencies of assessment failures necessary to arrive at modifications that might not be ideal but are considered *tolerable*.

Despite the differences between SSM and product line engineering, both approaches produce solutions with variability as specified by a process. This major similarity allows SSM to exploit a new argumentation technology for product lines, discussed in the following section.

6.2.2 Product Line Assurance Cases

In a recent publication, Habli and Kelly [22] investigate the application of assurance cases to SPLE. The premise of their work is that SPLE might be beneficial for generating safety-critical software. The use of assurance cases for safety (i.e., safety cases) is a common and sometimes mandatory practice [17] to demonstrate that a software system is adequately safe. As a result, a safety case for all variations produced by the product line is a valuable and sometimes necessary asset; however, constructing an assurance case for each variation from scratch or in an ad hoc manner could undermine the productivity benefits that a product line provides. Instead, SPLE should support construction of an assurance case for each variation.

The intuition of assurance cases for product lines is that since SPLE produces software with common architectures, software components, failure modes, risk mitigation approaches, and so on, the assurance case for each variation will also share a similar form of argumentation. As there are reusable software components used to generate software in a product line, there are also reusable argument structures that can be similarly composed.

To provide an assurance case for any variation produced by a product line, Habli and Kelly define a single assurance case that uses a special notation to indicate points of variability. As a product line generates a variation, decisions are made corresponding to these points of variability. Once a decision is made, a corresponding argument is selected and concretized. In this manner, the software product line assurance case acts as a blueprint that is concretized and pruned based on decisions made during the product line engineering process.

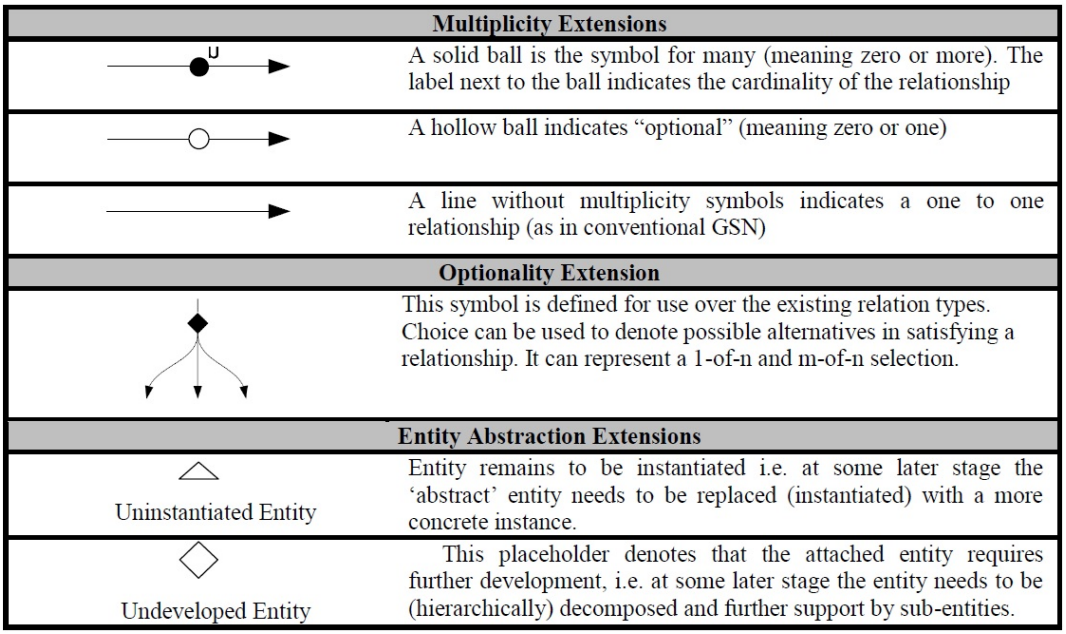


Figure 6.3: GSN pattern notation elements — figure taken from Habli and Kelly [22]

Habli and Kelly adopt GSN pattern extensions to facilitate assurance cases for product lines. GSN pattern extensions are described in Chapter 2, but for ease of reference, these extensions are illustrated in Figure 6.3. Pattern extensions provide two types of abstraction [22]:

Structural Abstraction: Represented by the multiplicity and optionality extensions in Figure 6.3, these extensions express n-ary, optional, or alternative relationships between argument elements.

Entity Abstraction: Represented by entity abstraction extensions in Figure 6.3, these extensions are attached to argument elements to indicate the argument element is either not instantiated or requires further development. Entity abstraction allows generalization and specialization of argument elements.

The structural and entity abstraction provided by these extensions allow the argument to capture and restrict variability in the argument. A key difference with the traditional use of GSN pattern extensions is that within a product line assurance case, every multiplicity or optionality element

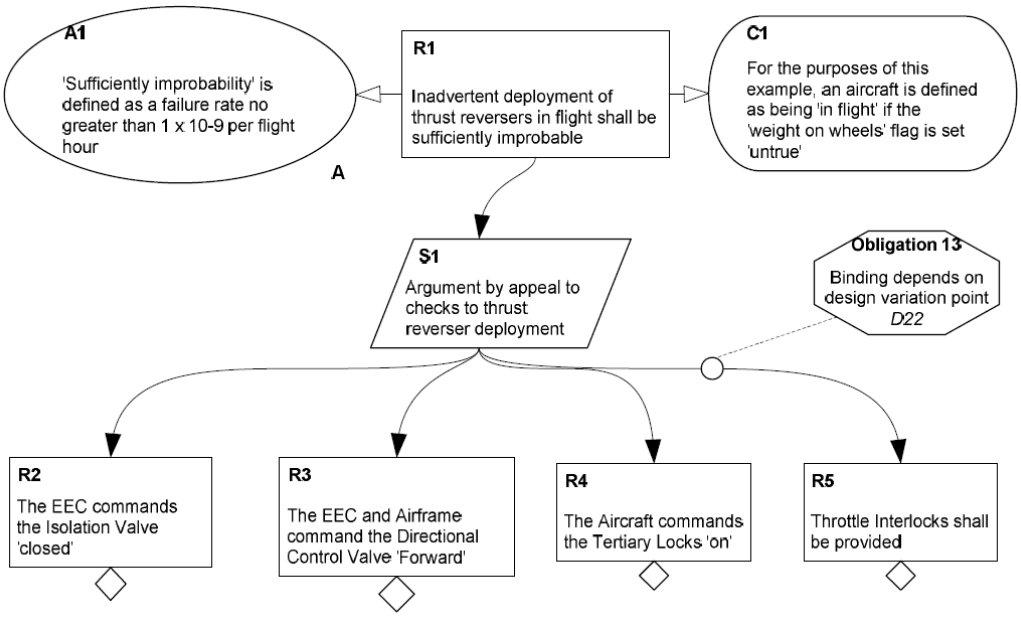


Figure 6.4: Product line argumentation example — figure taken from Habli and Kelly [22]

has an attached GSN *obligation* element (depicted as an octagon symbol) to indicate the criteria by which optional arguments or alternatives are chosen[1].

Figure 6.4, taken from Habli and Kelly [22], demonstrates the use of GSN pattern extensions to support variability in a product line assurance case. This example argument fragment is used to demonstrate the improbability of an inadvertent deployment of reverse thrust for an aircraft in flight. Goal R5, '*Throttle interlocks shall be provided*', is optional, as indicated by the optional multiplicity element (depicted as a hollow circle). Some manufactures do not require throttle interlocks; hence, the inclusion of goal R5 is dependent upon the kind of engine used. Obligation 13 is used to indicate the dependency under which the inclusion of R5 is based, referring to the dependency as variation point '*D22*'.

[1]Obligation elements are not typically used within argument patterns. Obligation elements are extensions to the GSN pattern notation and are therefore not shown in Figure 6.3.

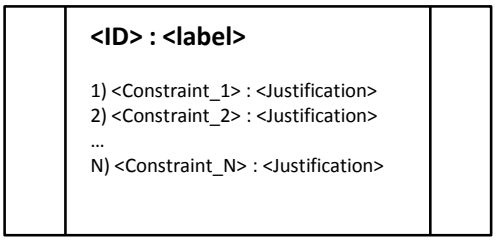


Figure 6.5: Argument guard element

6.3 Argument Guards

The unknowns and uncertainties of SOUP modification (see Chapter 1) make the construction of an SSM assurance case difficult. Specifically, evidence to support an argument might depend on the characteristics of the software, which are not known during the development of the assurance case. Instead of abandoning argument construction, developers can base the argument on *hypothesized evidence*, i.e., evidence assumed to be available.

When the software is modified, assessments are required to determine if hypothesized evidence is actually available. The purpose of the assessment activities within the SSM process is to validate hypothesized evidence. Failure to generate necessary evidence means that certain portions of the argument cannot be used to justify the fit-for-use claim. To express a summary of necessary evidence that affects the validity of argument structures, selection argumentation introduces the concept of *guarded arguments*.

Guarded arguments are similar in concept to *guarded commands* [64]; however, instead of guarding the execution of a command, an argument guard restricts inclusion of components of an argument within an assurance case instance. A guard is depicted as a GSN criterion element (Figure 6.5), further referred to as a guard element. Guard elements are connected by a solid line to restricted argument elements. Typically, a guard indicates evidence that must be generated to support claims found throughout the assurance case (in both the fitness argument and success argument). For example, the use of a sub-argument might be restricted by:

- Evidence corresponding directly to the sub-argument, i.e., found within a given sub-argument.

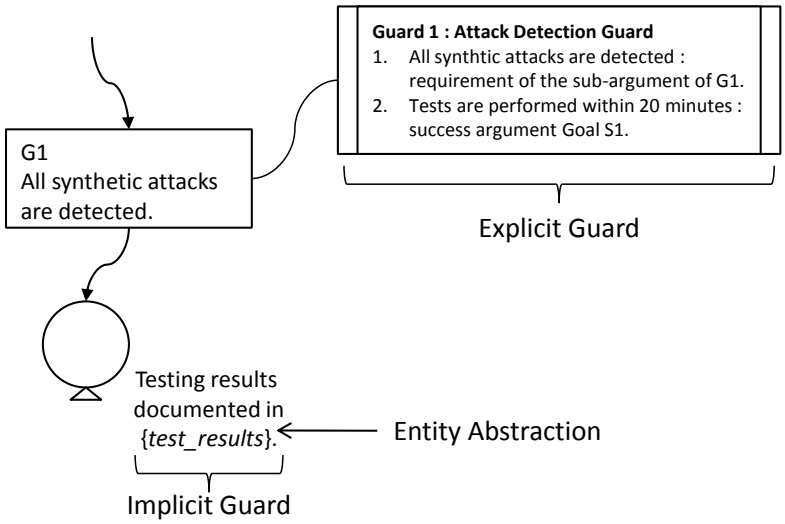


Figure 6.6: Example of explicit and implicit argument guards

- Evidence corresponding to competing constraints such as correctness and efficiency (Chapter 4).
- Evidence corresponding to confidence arguments (Chapter 5).
- Evidence corresponding to development constraints found within the success argument (Chapter 3).

6.3.1 Explicit vs. Implicit Guards

Entity abstraction used by product line argumentation can also serve as a form of *implicit argument guard*. Entity abstraction indicates some entity (e.g., data used as evidence within the argument) must be generated and have a specific form. Failure to generate the necessary data would imply the associated argument is not valid. We therefore make the distinction between implicit and explicit argument guards, illustrated in Figure 6.6:

Definition 6.3. ***Implicit Argument Guards***: *The use of product line entity abstraction alone to restrict the use of an argument entity in the generation of an assurance case instance.*

Definition 6.4. ***Explicit Argument Guards***: *The explicit documentation of restrictions affecting the use of an argument element in the generation of an assurance case instance. Explicit guards are documented using guard elements, shown in Figure 6.5.*

The key benefits of using explicit guards instead of relying entirely on entity abstraction (i.e., implicit guards) are:

- Explicit guards highlight the exact constraints restricting the use of an argument element in an assurance case instance.
- Explicit guards can express crosscutting constraints corresponding to other arguments spread through the SSM fitness and success arguments.
- Explicit guards can be placed at any position in the argument to better clarify how an argument is restricted, instead of relying on reviewers to traverse the argument to find all implicit guards.
- Explicit guards always document argument restrictions. Entity abstraction can be used as a placeholder for identifiers, file locations, etc. that are not known a priori. These uses of entity abstraction do not necessarily restrict the use the argument.
- Explicit guards can document any kind of restriction, including restrictions based on evidence that must be validated and not generated. Entity abstraction, however, only provides restrictions when the exact form of some entity (e.g., evidence) will have to be derived later.

While the use of explicit guards over implicit guards is at the discretion of the argument developers, for simplicity, further discussion of guards is with respect to explicit guards. The same general semantics and mechanics for explicit guards also apply to implicit guards.

6.3.2 Guard Format

An explicit guard is documenting using a guard element illustrated in Figure 6.5. Each guard element specifies one or more validation criteria, referred to as a constraint. To allow auditors of the

argument to understand the rationale for all constraints within a guard, each constraint has one or more corresponding justifications, in the following form:

$$< constraint\_identifier >< constraint >: < justifications >$$

In many cases, the justification for a constraint is found elsewhere in the assurance case (i.e., the constraint corresponds to evidence in support of some goal in the assurance case). In this case, justification is documented as a reference to applicable goals, contexts, evidence, etc.

For example, the guard in Figure 6.6 describes a restriction on a security claim. In this case, not only must testing be performed to validate the associated claim, but testing must also terminate within a time constraint. The justification for the time constraint is found within the success argument. When selecting a modification approach, crosscutting competing constraints/restrictions such as these are often not immediately apparent. By placing constraints within a guard, the argument more easily conveys when modification techniques can be acceptably applied. The justification for why these constraints are necessary is found elsewhere within the assurance case as referenced by each constraint's justification. Further illustration of the use of guards is described in Section 6.6.

6.3.3 Guard Semantics

A guard, G, is *valid* if each constraint, C, within the guard is valid (i.e., each constraint is met):

$$Valid(G) \iff Valid(C_1) \wedge ... \wedge Valid(C_n)$$

A guard is placed in relation to an argument element, E, (i.e., a goal, context, justification, assumption, etc.) to restrict the use of E in selecting an assurance case instance. An argument element is selected for use within an assurance case instance if is *permissible* to use (denoted $Perm(E)$):

$$Perm(E) \iff (\forall G : GuardsOf(E) \bullet Valid(G)) \wedge PermPath(E)$$

As indicated in the above predicate, valid guards directly associated within an argument element ($GuardsOf(E)$) are a necessary but not always a sufficient condition for $Perm(E)$. If E is a goal element, guards might be present within the sub-argument for that goal. Hence, for $Perm(E)$ to be true, a path through the sub-argument of E must exist where all guards are valid. This path is referred to as a *permissible path* (denoted $PermPath$):

$$PermPath(E) \iff \exists P : Paths(E) \bullet (\forall G : PathGuards(P) \bullet Valid(G))$$

A guard only restricts the use of an argument element in constructing an assurance case instance. Ultimate selection of an argument element might depend upon one or more decision models (discussed in Section 6.4). Thus, we use the term *permissible* because the element's use might be predicated on a decision process.

A *path* through the argument is therefore a single and complete traversal of the argument in which all decisions between alternatives have been resolved. The set of all argument paths stemming from specific argument element E is denoted $Paths(E)$. If E is the top-level goal, $Paths(E)$ is the set of all assurance case instances that can be constructed. $PathGuards(P)$ refers to all guards that restrict the use of argument elements on a given path, P.

Constraints within each guard correspond to assessment activities performed during the SSM process. If during the SSM process the constraints within the guard cannot be validated, the guard is said to be *invalidated*, triggering the decision refinement mechanics discussed in Section 6.4.

6.4 Decision and Refinement Models

By using product line argument notation within a fitness argument, the argument can specify choices between alternatives within the argument elements (e.g., alternative goals, contexts, evidence, etc.). These alternative argument structures can correspond to methods for analyzing and modifying the SOUP in SSM; hence, these alternatives also correspond to hypotheses about the manner by which a SOUP modification is generated.

Within SSM, decision processes can be more complex than originally intended in product line engineering. Decisions within an SSM argument do not merely codify the need for engineers to choose between alternatives, and relationships/conditions affecting the choice. Rather they are more mechanized and further indicate how decisions are readdressed (i.e., refined) as a previous choice is invalidated. At each decision point, an obligation element is attached to model the decision and refinement process.

The decision process can be modeled in any number of ways, such as:

A maximizing operation: The maximum number of alternatives under a decision point should be chosen.

A fitness operation: A fitness function specifies a set of characteristics that must be maximized. The alternative with the highest fitness (the maximized characteristics) is chosen.

Probabilistic models: Selection between alternatives is determined by calculating a probability of success based on characteristics of the SOUP being modified.

Predefined preference: A preference between alternatives is known a priori. Selection is codified as a linear progression between alternatives to find the first valid option.

Regardless of how decisions are made, the model must support refinement of the decision if a choice is later invalidated. In SSM, a decision is invalidated if the argument guards associated with a selection are later invalidated. To address the possibility of decision invalidation, each decision model should preserve the state of the previous decision, and have an explicit method by which a new decision is made. For example:

- Decisions based on maximizing operations can choose to drop/ignore invalidated choices, and simply use the remaining validated options.
- Decisions based on fitness functions can choose the next most fit alternative.

- Probabilistic models can update probability distributions based on observed events, e.g., using a Bayesian probability, and then choose the next alternative based on the updated probability of success.

- In a predefined approach, the next option is selected in the chain of preferences.

The field of decision theory is vast, and many decision models are potentially useful within SSM. We do not restrict which models are applied, and further exploration of various decision models is outside the scope of this dissertation; however, the next section provides an example decision model based on finite-state machines for illustration.

6.4.1 Modeling Decisions using Finite-State Machines

To provide an illustration of a decision and refinement model for discussion, this section presents the mechanics of an example model. In many cases, an SSM decision process can be described is a progressive (linear) selection between a predefined preference of alternatives. As a preferred alternative is invalidated, the manner in which the alternative was invalidated governs selection of the next preferred option.

This style of decision process is amenable to modeling with a finite-state machine (FSM), also referred to as finite-state automata. The intuition is that each selection between argument alternatives is a *state*. When a choice is later invalidated, the invalidation triggers a transition to another state, where a new choice is made.

Mathematically, an FSM is expressed as the 5-tuple $(\Sigma, Q, q_0, \delta, F)$ where these elements have the following meanings:

- Σ is the set of input symbols (i.e., the alphabet of inputs that trigger state transitions).

- Q is the set of states within the FSM.

- $q_0 \in Q$ represents the start state of the FSM (i.e., the state from which transitions in the FSM begin).

- δ represents the state transition function of the FSM, expressed as $\delta : Q \times \Sigma \rightarrow Q$ for deterministic FSMs and $\delta : Q \times \Sigma \rightarrow P(Q)$ for non-deterministic FSMs.

- F represents the set of final states (accepting states) of the FSM.

Within selection argumentation, an initial preference between argument options is the FSM start state. If the initial choice is latter invalidated through assessment, the precise nature of the failure (further referred to as a *failure condition*) can be used as an input symbol to the FSM, triggering a transition to another state (another argument option) based on the conditions of the failure. The concept of an accepting state corresponds to any selected alternative that is not invalidated, i.e., each state is a final/accepting state. Acceptance is triggered when no additional assessments are preformed to validate the choice, i.e., all guards associated with the selection have been validated.

In traditional FSMs, when an input symbol is not recognized (the FSM does not specify a transition for the input symbol), the FSM is said to be in an *undefined state*. In our FSM decision model, if a choice (i.e., a state) is invalided by means not documented explicitly in the transition function, the FSM transitions to an implicit rejection state. The mechanics of how rejection is dealt with are further discussed in subsequent sections.

6.4.2 FSM Illustration

Figure 6.7 illustrates an argument excerpt in which a decision is required between three sub-argument alternatives. In this example, each sub-argument corresponds to methods for arguing adequate mitigation of an attack class. The decision model for selection between these alternatives is found in document D1 (referenced within the obligation O1) and is modeled as a finite-state machine. For simplicity of the illustration, no detailed argument structure is given for each alternative, but we assume each alternative has associated argument guards.

The graphical representation for the decision process FSM is illustrated in Figure 6.8. For illustration purposes, the precise failure conditions are not given and are important for the example. The key concept of the illustration is that each alternative has associated guards, and those guards

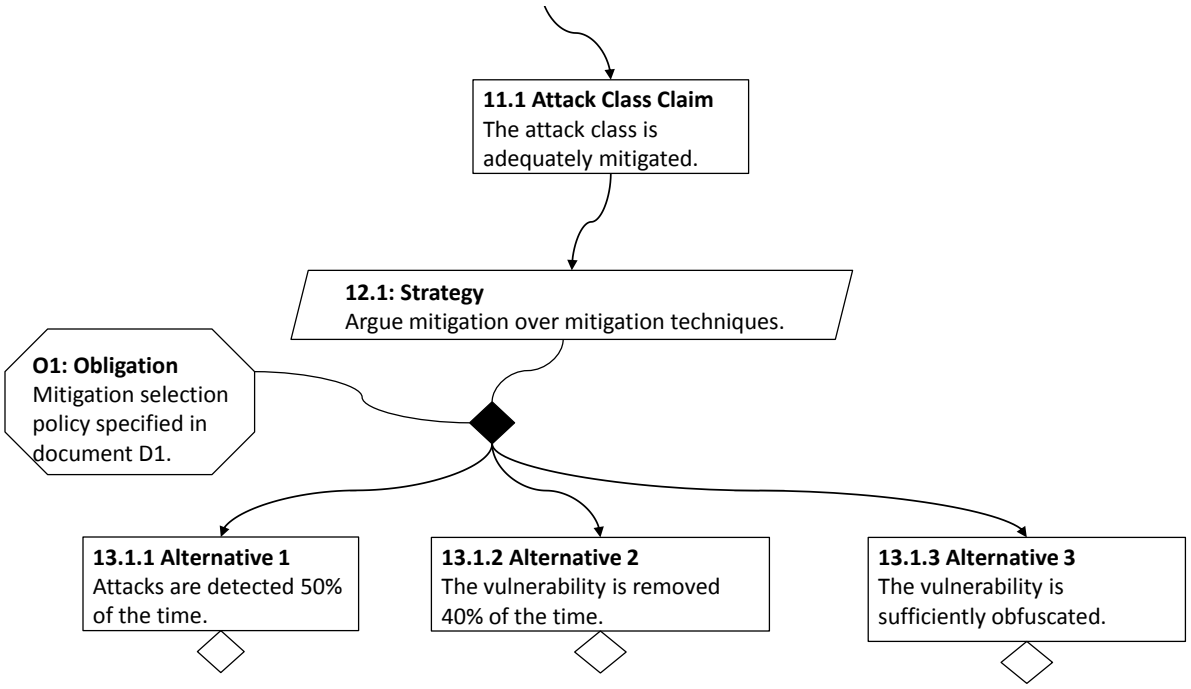


Figure 6.7: Argument decision example

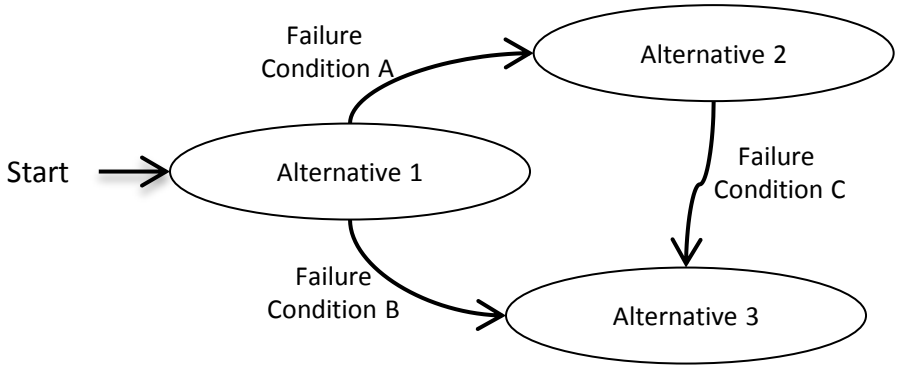


Figure 6.8: Example finite-state machine decision process for Figure 6.7

can be invalidated. While predefined preference between approaches can be a simple linear chain of states, this diagram illustrates a slightly more complicated decision model. Under a certain chain of failure conditions, the choice between alternatives would proceed in a linear sequence, i.e., from alternative_1, then alternative_2 and finally alternative_3; however, it is possible to bypass alternative_2 altogether under the right failure conditions, and skip directly to alternative_3.

State machines are also amenable to representation as transition tables, shown in Table 6.1.

Failure Condition	Current Choice		
	Alternative 1	Alternative 2	Alternative 3
Failure Condition A	Alternative 2	-	-
Failure Condition B	Alternative 3	-	-
Failure Condition C	-	Alternative 3	-

Table 6.1: Transition table example of the finite-state machine diagram illustrated in Figure 6.8

Given a state and a transition, a transition table indicates the next state. Transition tables are easily represented in text-based forms obviating the need for graphical tools, and facilitating documentation of these decision processes within the argument itself.

6.4.3 Decision Mechanics and Conditional Tolerability

Regardless of how decisions are modeled, the decision mechanics within selection argumentation behave in the same manner. When a decision is made, guards associated with the select argument are validated. If any guards are not valid, the evaluated failure condition guides a traversal back up through the argument to the nearest decision point. The decision model at that point is consulted and the next choice is made.

If the nearest decision model cannot reach another decision, the failure condition continues to propagate hierarchically up through the argument to the next decision point. For example, using the transition table described in Table 6.1, if a failure condition occurs for any alternative that is not accounted for in the transition table, all alternatives are rejected. Since no decision can be made, the failure condition continues to traverse up through the argument to next decision point. If no valid alternatives can be found through back propagation, the entire argument is considered invalid. Within a fully instantiated SSM process, invalidation of the entire argument would result in a modification error/failure message (see Chapter 3).

A consequence of making other choices as preferred choices are invalidated is that SSM modifications can have a range of tolerability, e.g., one modification might be preferred initially over another, yet both can be considered tolerable for use in certain scenarios. The decision and

refinement process can be used to dictate how to "back off" progressively until a tolerable solution is found. We refer to this concept as *conditional tolerability*.

Definition 6.5. ***Conditional Tolerability***: *The conditions for which a chosen configuration or strategy for analysis or modification is considered tolerable. A conditional tolerability defines a dependency chain of necessary conditions (i.e., necessary failures of other approaches) that must occur in order for a given approach to be used and tolerated. Failure to meet these necessary conditions would mean the approach is not tolerated.*

6.5 Guidelines for Application and Interpretation

The intent of selection argumentation is essentially to describe an argument development process within a fitness argument. The SSM process, and those manually reviewing the argument, would then proceed by traversing the documented development process.

A challenge of this approach is that a fitness argument contains separate argument branches for the efficacy of establishing the desired property (e.g., security or safety), and the ability of the modification to meet stakeholder-defined constraints (e.g., correctness, efficiency, confidence). These argument branches are not independent. If branches of the argument are dependent on each other and also express a complex decision process, there is no clear indication of the order in which branches should be traversed. Further, the interaction becomes difficult to understand. To provide a simpler and comprehensible argument selection mechanics, we therefore choose a single argument branch where selection argumentation is applied. Selection argumentation mechanics are then interpreted by traversing the chosen argument branch.

We observe that the primary purpose of constraints within the SSM assurance case is to place restrictions on the branch of argument that justifies the *efficacy* of the modification, i.e., the argument that demonstrates desired dependability properties are present in the modified software. For example, in the security fitness argument illustrated in Chapter 4, the efficacy branch would be the security requirements branch.

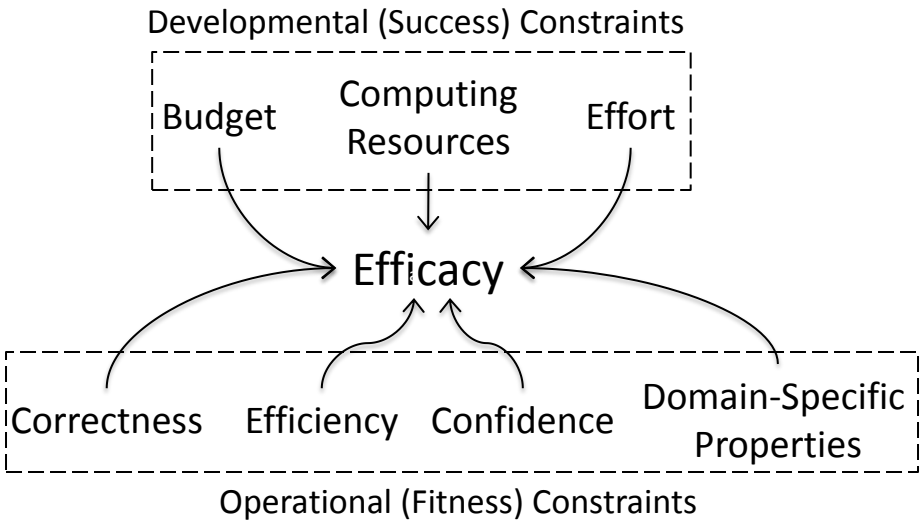


Figure 6.9: Constraints of modification efficacy

Arguments in all other branches of the fitness argument, and all arguments within the success argument, serve to restrict choices corresponding to modification efficacy, illustrated in Figure 6.9. We therefore stipulate that the efficacy branch of the fitness argument is where selection argumentation should be primarily applied. Interpretation of the selection argumentation mechanics proceeds by traversing the efficacy branch and resolving selection argumentation concepts (i.e., making decisions, validating guards and refining decisions) as they are encountered.

The general guideline for focusing the use of selection argumentation is to restrict argument guards and intricate decision models to the efficacy branch. As arguments are constructed for the efficacy branch, if any component of sub-argument is dependent on evidence that must be generated or assessed (from any part of the success or fitness argument) a guard is placed in the efficacy branch. Product line argumentation is applied to express variability and decision models are applied to express preference between alternatives and decision refinement procedures.

All other areas of the fitness argument might lack data or have some variability in a purely product line sense, in which case product line argumentation is still permissible. For example, a correctness argument based on testing a modification will need to be produced when the modification is generated. Any associated correctness argument will therefore not have the testing results a priori. The missing evidence would be indicated in the argument using entity abstraction. Somewhere

in the efficacy branch, a guard is placed to restrict selection based on generating sufficient testing evidence to suggest the approach is adequately correct.

The only exception to the use of guards is for cases for which evidence cannot be generated until the modification is completed. For example, a correctness claim could be made about the whole modification, not just one of its components. If the claim is based on testing the completed modification, testing data will have to be generated during modification finalization. Missing evidence corresponding to finalization activities such as these does not serve to restrict decision in the efficacy branch. Rather, the primary purpose is to determine if the argument as a whole is sufficient. To highlight evidence of this form, the use of guards is permitted anywhere in the fitness argument.

6.5.1 Limitations and Assumptions

Figure 6.9 intentionally belies any complex relationships between constraints. In principle, constraints can constrain other constraints. For example, an analysis to validate correctness might be constrained by confidence that the approach will produce reliable results. Further, the analysis can be constrained by the time allowed for generating data. The application and interpretation of selection argumentation is based on the assumption that all constraints ultimately serve to restrict efficacy decisions, either directly or indirectly. Additionally, by focusing the use of selection argumentation within the efficacy branch, there is an assumption that no other branches of the fitness argument will require complex decision processes, i.e., all SSM decision processes are based on efficacy.

These assumptions and uses of selection argumentation limit where variability is allowed within the fitness argument. Specifically, variability is not permitted in how correctness, efficiency, confidence, or any other constraints are argued. We make this stipulation to promote simplicity of SSM assurance cases, and to focus and simplify the content of this dissertation. Selection argumentation can be expanded to allow for variability in other areas of the argument through more

complex guard mechanics; however, investigation into extending the use of selection argumentation for more complex decision processes is left for future research.

6.5.2 Relationship to the SSM Process

The SSM assurance case is interpreted by traversing the fitness argument through the efficacy branch. Decision models indicate how choices between alternative argument structures are made, and guards indicate all evidence that must be generated and validated for an alternative to be acceptable. Failure to validate the constraints within a guard results in failure propagation up through the argument (towards the top-level claim) to the nearest decision model. Reviewers must also consult the success argument for global resource consumption constraints, in addition to constraints that affect individual decisions. For example, a timeout condition on the SSM process is part of the success argument, but does not directly influence any individual decision.

Traversal continues until all decisions are processed, or the process is terminated based on exceeding acceptable levels of resource consumption (as specified in the success argument). Whatever path has been traversed constitutes an assurance case instance. A final review of the argument is necessary to generate any remaining missing evidence, and to verify that the argument is not missing any necessary components.

The SSM process is an instantiation of this kind of argument traversal; however, there is no direct mapping from the argument mechanics to the process components of SSM. Manual review of the argument is necessary to determine how the SSM process should be implemented. Ultimately, how the SSM process instantiates the mechanics of the SSM assurance case are subjective; however, some general guidelines for mapping the argument mechanics to an SSM process are provided below.

Preprocessing: Preprocessing allows for the optimization of the SSM process by generating or validating data that will be useful for the remainder of the SSM process. SSM developers should examine all decisions and guards within the fitness argument to find any data that could be generated

or assessed in preprocessing. For example, a guard that is unconditionally assessed (i.e., the guard is not associated with any decision) represents a fundamental restriction on the entire fitness argument and it should be assessed in preprocessing. If the guard is invalid, no modification will be valid; hence, there is no reason to continue with the SSM process. Similarly, if a large number of alternatives are guarded based on the same constraint, the SSM process might be optimized by performing the assessment in preprocessing.

Speculative Analysis and Modification Synthesis: Decision models within the argument structure correspond to strategy selection in both the speculative analysis and modification synthesis phases. Each alternative that can be chosen by a decision process is an analysis or modification approach. The exact analysis or modification approaches to perform are expressed within the semantics of all claims associated with each argument alternative. Guards associated with alternatives correspond to assessment metrics. The distinction between speculative analysis and modification synthesis is based on if a modification must be synthesized to perform the assessments or not. Assessment summaries in the SSM process express either that associated guards have been validated, or the guard constraints that were invalidated.

Termination: The SSM process will terminate when either an explicit or implicit termination condition is reached. An implicit termination condition corresponds to fully generating an assurance case instance, i.e., processing all decisions and arriving at a complete traversal of the argument. An explicit termination condition is the result of not being able to find any acceptable alternatives or if the resources consumed by the modification process (as specified in the success argument) exceed some defined limit. The former case is a failure to produce an acceptable modification, resulting in the SSM process producing an error message. In all other cases, the SSM finalization process is initiated.

Finalization: Finalization involves a last assessment of an assurance case instance. Assessment of the assurance case instance involves verifying that the argument is complete by verifying

necessary arguments are present and that necessary evidence has been generated. Some evidence cannot be generated until finalization. All guards with constraints that can only be verified at finalization are assessed during finalization. Failure to validate the assurance case instance results in SSM producing an error message. Otherwise, the modification was successfully generated.

6.6 Illustrative Example

Figure 6.10 illustrates the combined use of the selection argumentation concepts within a fitness argument for security. This example is based upon our first target for SSM evaluation, described in Chapter 7. In this example, a set of modification techniques are applied to individual functions within a binary program (binary SOUP) to mitigate security risks. The selected argument justifies that the modified software adequately mitigates a specific set of attacks. For the sake of illustration, the exact nature of the attack class and the type of mitigation techniques applied are not important; however, a more detailed description of this example is provided in Chapter 8.

In this example, decision and refinement models are relatively simple and are described in plain text within each obligation element. The first decision, specified in obligation O1, indicates a maximizing decision process associated with a multiplicity element (a black dot). A multiplicity element indicates the argument structure below will repeat. In this case, the decision process in obligation O1 specifies the number of argument repetitions corresponds to the number of functions that are mitigated from attack. While ideally all functions will adequately mitigate an attack, some might fail to be fully mitigated. The decision process at Obligation O1 specifies a maximizing operation, and no more complex refinement activities. As such, if a function fails to be mitigated, the failure does not invalidate the entire argument.

To address the risk that too few functions will be adequately mitigated, goal 13.3 argues that a minimum number of functions are mitigated. Exactly how many functions are mitigated cannot be known prior to generating the modification; hence, guard G2 is used to indicate the dependency of

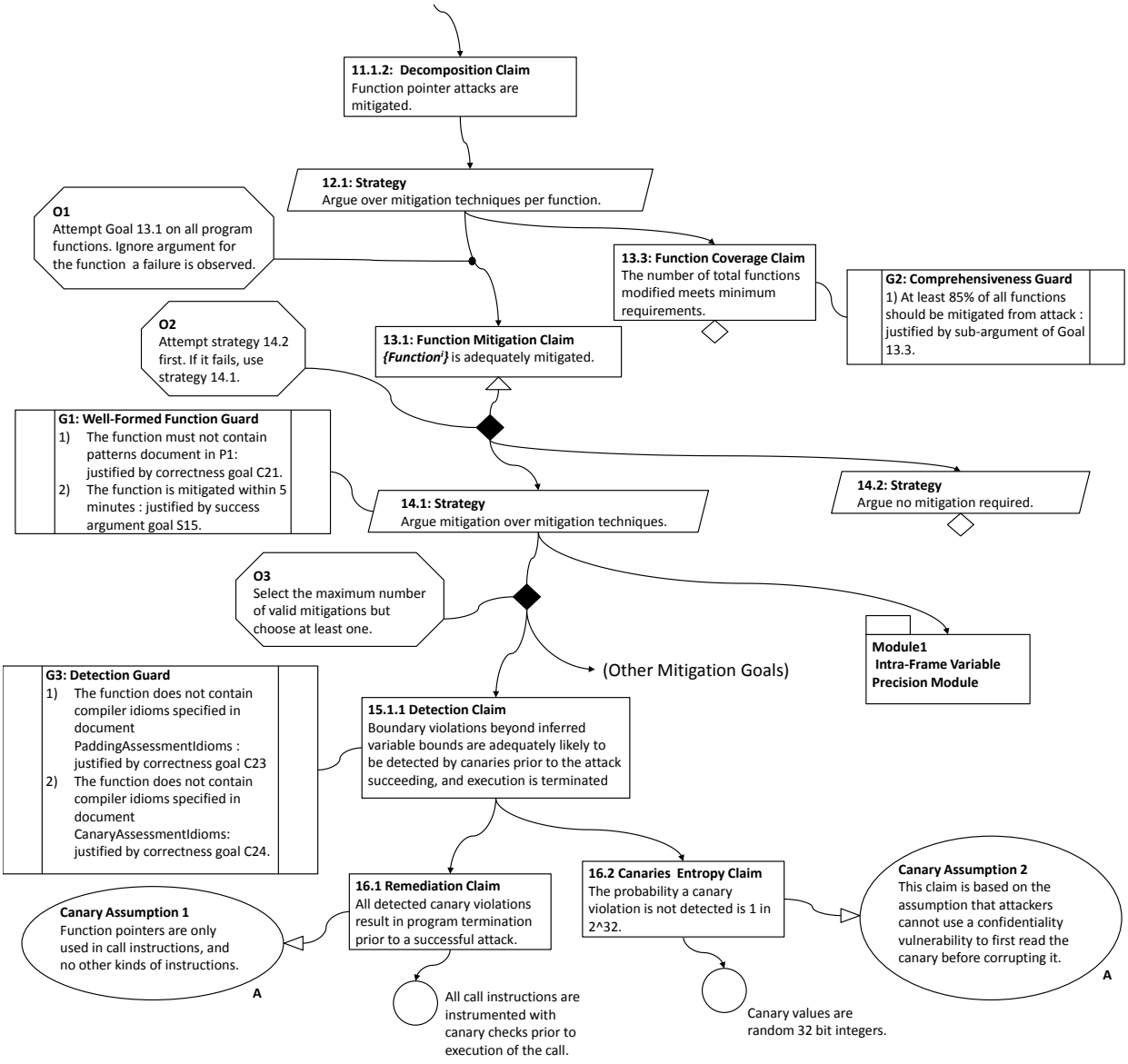


Figure 6.10: Example application of selection argumentation within a fitness argument

goal 13.3 on evidence about the completed modification (i.e., evidence that will be generated and assessed in the finalization phase of the SSM process).

Each function can mitigate the attack using one of two general strategies:

1. Sufficient evidence exists that the function not at risk and does not require mitigation (strategy 14.2).

2. The function is considered at risk and attack mitigation techniques are applied (strategy 14.1).

The next decision point, document in obligation O2, describes the decision model to select between these two strategies. In this case, the preference is to attempt to argue that no mitigation is necessary (strategy 14.2). If strategy 14.2 is not validated, then strategy 14.1 should be attempted. For simplicity, the sub-arguments and further details corresponding to strategy 14.2 are omitted; however, it is assumed that strategy 14.2 uses guards within its sub-argument.

If strategy 14.2 is selected, the function in question is modified using a set of mitigation techniques. In this example, all mitigation approaches rely on a set common assumptions and guard G2 makes these key assumptions explicit as constraints restricting the use of strategy 14.1:

1. All functions must have a common and expected form in terms of how the function accesses memory, the instructions the function uses and the overall patterns of instructions. Certain function forms are known to be problematic when modified, i.e., functions of the specific form can negatively influence the correctness of the modified software.

2. Each function must be modified within a specific time limit (5 minutes). In some cases, mitigation techniques require additional assessments and testing to derive necessary evidence to support the argument. The SSM success argument is used to constrain development resource consumption, yet these constraints have a direct impact on the choice of mitigation techniques.

The last decision point, documented in obligation O3, indicates how mitigation techniques are selected. The selection policy is to choose as many mitigation techniques that are valid, but choose at least one. For the purposes of illustration, we focus on a particular mitigation technique for detecting attacks (goal 15.1.1). Use of the detection mitigation technique is guarded by the constraints specified in guard G3. Much like guard G2, the constraints in G3 indicate specific patterns of instructions that can negatively affect correctness. If these patterns are found, goal 15.1.1 cannot be used within an assurance case instance.

Chapter 7

Case Study: Exploring the SSM Process

This chapter presents the the first in a series of three case studies to evaluate and examine the SSM model. For all three studies, we sought to test the hypothesis that the SSM model is both feasible and practical by applying the model to SOUP modifications for security. *Feasibility* refers to the realistic/reasonable application of the SSM approach and *practicality* refers to the ability of the approach to provide real benefits for engineers of software modifications for security. Over the course of each case study, the concepts and guiding principles of SSM were discovered and refined. Subsequent studies exercise/illustrate and further refine discovered principles.

7.1 Case Study Overview and Goals

This chapter presents a case study designed to answer the research questions:

What are the general motivation, form, rationale, and potential utility of SSM?

Since each study performed in this dissertations builds upon discoveries and observations in sequence, we did not begin this study with a complete concept/understanding of the SSM model. Instead, we began with a basic security problem, and our solution to that problem yielded observations leading to the concept and mechanics of the SSM process. We first present the SOUP

security problem and our modification technique to address this problem, referred to as Stack Layout Transformation (SLX). A primary result of this study was the identification of the need for explicit models of acceptability in SSM model (i.e., the SSM assurance case), further studied in the next chapter.

The goal of this case study is to evaluate the feasibility and practicality of the SSM process by investigating the process mechanics of the SSM approach found in SLX. Specifically, this case study addresses the following research questions:

Question 1 Is the SSM process feasible?

- Can the principles of the SSM process be implemented to solve real-world security issues?

Question 2 Is SSM process practical?

- Can the SSM process be used to generate software modifications that achieve stakeholder-defined levels security within defined constraints?
- Does the process present challenges that would affect or limit how or when SSM can be applied?

Question 3 Is the model of SSM complete?

- Are there limitations not addressed by the SSM model?

7.2 Targeted Vulnerability

In this case study, software modifications are designed to protect against stack-based buffer overflow attacks. Despite efforts by both researchers and practitioners, stack-based buffer overflow attacks remain a common and serious threat [65]. Stack-based buffer overflow can be avoided if the software is engineered with this risk in mind, such as using memory safe programming languages (e.g., Java), and memory safe libraries (e.g., libsafe [66, 67]). Overflows remain a considerable security risk

today since not all engineers consider security or buffer overflow vulnerabilities during development, and therefore stack-based buffer overflow represent a relevant concern for users of SOUP.

If the program source code were available, protection mechanisms against stack-based buffer overflow would be relatively simple to apply, because high-level program abstractions and semantics, e.g., variable location and type, are directly specified [68, 69, 70, 71]. Unfortunately, precise recovery of source-code level abstractions and semantics is not feasible once a program is compiled into its binary form [12]. As a result, existing methods to protect against buffer overflow in binary programs typically apply only limited (but sound) protection mechanisms, e.g., by randomizing the base address of the stack region or by transforming an entire stack frame [72, 73, 74, 75]. Such coarse-grained protections do not protect against buffer overflows between local variables and non-control data attacks [76].

Rather than rejecting potentially useful but unsound protection mechanisms because of the associated risks, our approach is to apply SSM in order to allow stakeholders to take considered risks to protect binary programs against stack-based buffer overflow attacks. Our SSM instantiation, referred to as *Stack Layout Transformation* (SLX) [77, 78], adds properties to the stack frame of each individual function in a program at the level of individual variables. SLX establishes defenses against buffer overflow by making alterations to the layout of each function's stack frame through the application of a combination of protection mechanisms.

7.3 SLX Overview

SLX applies transformations to individual function stack frames of a binary program by:

1. randomizing the order of local variables within the frame,
2. adding random-length padding between variables, and
3. placing *canaries* (random values that are checked periodically at run-time to detect overflows) between variables.

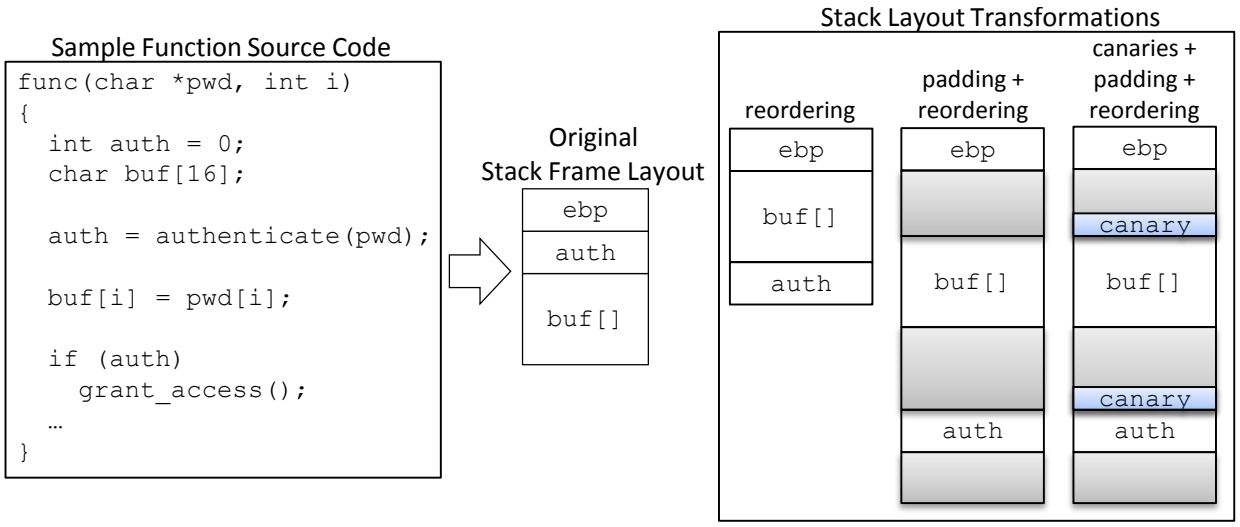


Figure 7.1: Stack layout transformations

SLX individually applies these modifications function-by-function. For defense in depth, SLX attempts to compose all three modifications for each function's stack frame (see Figure 7.1).

This choice of transformations was based on similar approaches applied to source programs or minimally applied to binary programs [70, 79, 69, 71, 72, 73] and Address Space Layout Randomization (ASLR) [75]. Reordering and padding variables diversifies stack layouts, thereby perturbing the predictable layout of data upon which an attacker might rely. Padding can also serve to allow for continued execution during an attack, i.e., attacks corrupting only the padded memory are essentially inert. Canaries also transform the layout of the stack by adding "tripwires" allowing for run-time detection and remediation of some stack-based attacks. For this case study, the remediation policy for a detected overflow is program termination.

Implementing SLX modifications on binaries presents many challenges. High-level structural and semantic information needs to be recovered, including:

- The layout of variables in each stack frame.
- The instructions that access the stack have to be modified.

In addition, a determination needs to be made of:

- Which modifications can be applied safely.

- How to configure chosen modifications.

This information is not necessarily recoverable with complete accuracy from a binary program. Thus, SLX generates hypotheses for the necessary information, and then assesses and refines the hypotheses using the SSM process architecture. SLX is implemented based on the mechanics of the SSM process, i.e., SLX is based on predefined analysis techniques, modification techniques, and assessments.

7.4 Motivating Use Case

Before discussing the instantiation of SLX in detail, we emphasize that SSM is intended to produce modifications that are acceptable, practical and proactive (see Chapter 1). Of key importance is the practicality component. Engineers have to work with available methods that are cost effective to use and produce. Often these methods are imperfect, but can be acceptable within certain contexts.

The chosen transformations used in SLX do not represent new advances in the state of the art of buffer overflow protections; rather they are based largely on previously studied and applied approaches. Each of the chosen transformation has considerable risks in applying the method (especially applied to programs without source code), and each transformation also has its own security vulnerabilities, potentially allowing attacks to bypass the mechanism. Nevertheless, each transformation provides some security benefits. Additionally, each transformation is easy to generate and can be tuned to have acceptably low run-time overhead.

The novelty of SLX lies in the practical application of these existing transformation techniques to protecting binary programs using SSM principles. Readers who choose to consider other transformation methods are encouraged to consider the risks and unknowns associated with those methods, and how engineers would deal with those concerns. SLX represents an instance of how these problems can be addressed with SSM.

The variability in how each SLX transformation is applied and combined can also lead to debate and controversy. The presented example is neither the only nor a “definitive” SLX implementation.

SLX, as an instantiation of SSM, is used to distribute risks in a manner considered acceptable by a targeted set of stakeholders (see Chapter 1). The selected modifications, configurations, acceptability model and the approaches used to produce and assess hypotheses and modifications are all chosen based on concepts of acceptability specific to a target set of stakeholders.

To provide clarity of the intent of our instantiation of SLX, we target a set of hypothetical stakeholders derived from the following use case:

1. The stakeholders identify utility programs used throughout their organization as being at significant risk of stack-based buffer overflow. We use a set of Linux utilities (coreutils) as a representative set of the targeted programs.

2. The source code of the targeted utility programs is not available.

3. The stakeholders can easily derive the functionality that the targeted software provides and how to use the software, facilitating development of a test suite for these programs (either through manual or automated means). We use the regression tests provided for Linux core utilities as a representative test suite.

4. The stakeholders wish to apply SLX transformations, and to minimize the costs of engineering SLX using relatively simple analyses and heuristics in the implementation.

5. While stakeholders prefer a completely comprehensive modification, they are willing to accept the best effort modification that their implementation of SLX can provide.

7.5 SSM Process Decomposition

This section provides a detailed description of SLX by dividing the mechanics of SLX into the four primary SSM phases:

- Preprocessing

- Speculative Analysis

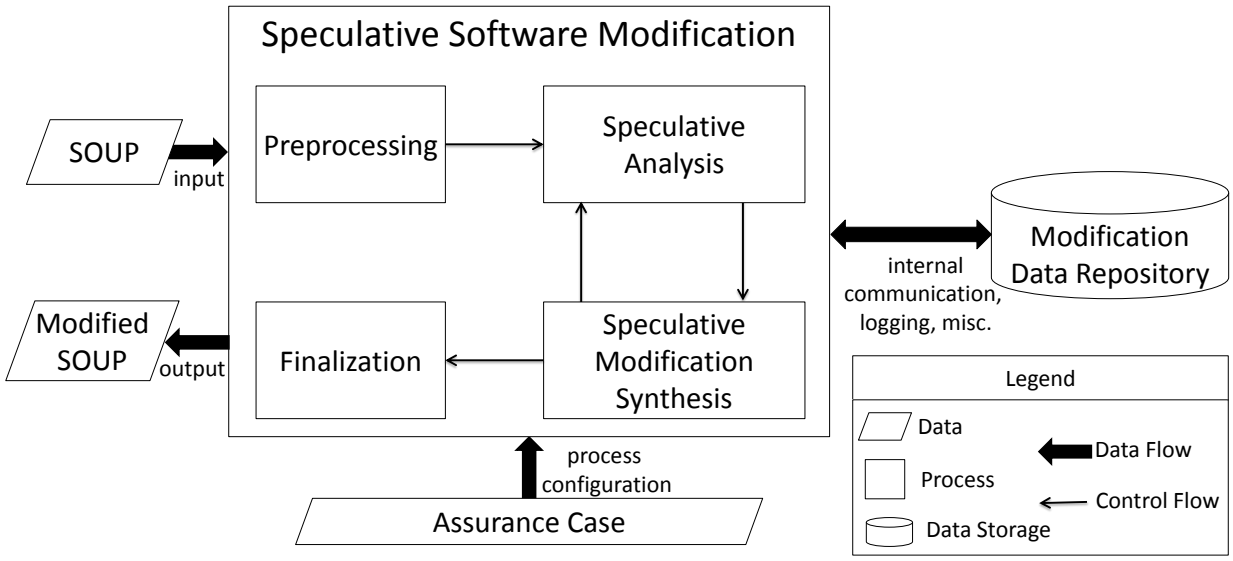


Figure 7.2: Detailed SSM process

- Speculative Modification Synthesis
- Finalization

These phases are illustrated and described in detail in Chapter 3; however, for ease of reference, the general SSM process model is repeated in Figure 7.2.

7.5.1 Preprocessing

Starting with a piece of SOUP, SLX preprocessing generates five items of information to be used by the remainder of the SLX process:

- the program disassembly,
- the control flow graph,
- identified function boundaries,
- an input test suite for the program, and

- execution coverage of the instructions in each function produced by testing the SOUP using the input test suite.

Inputs are used to guide dynamic analysis later in the SLX process. Preprocessing generates these inputs using a combination of concolic test case generation (similar in concept to klee [28]) to automatically generate inputs, and a database search to find appropriate pre-existing test inputs if they are available. The database search involves generating a signature of the program based on the strings found in the executable. If the signature closely matches a program for which a test suite is already available, the corresponding the suite is used.

Preprocessing also performs sanity checks to remove functions from consideration for modification that appear to be malformed and could result in bad SLX modifications. By malformed, we mean functions whose stack frames deviate from the expected format, probably as a result of disassembly recovery failures or unusual compiler optimizations and idioms.

7.5.2 Speculative Analysis

To effect an SLX modification, hypotheses for a binary program are generated function-by-function; hence, our initial implementation of SLX involved a repetition between speculative analysis and modification synthesis for each function. After each individual function is modified, the process will repeat for the next function until all functions have been processed (optimization of this process to modify numerous functions at once is discussed further in Section 7.6). An SLX hypothesis for a function contains:

1. the location and size of the variables in the stack frame (i.e., the stack-frame variable layout),
2. the instructions in the function that access variables in the stack (i.e., the instructions requiring modification in order for the program to execute with the modified stack layout) and,
3. the modifications to apply (i.e., the set of SLX modifications that are applicable to the subject function).

Algorithm 1 $HypothesisStrategySelection(DB, A(H(func)))$

Input: DB: database of analysis preprocessing results and
$A(H(func))$: assessment summary of a hypothesis for $func$

```
1: if A(H(func)) = NULL then
2:     func ← DB.getNextUnprocessedFunction()
3:     //termination condition check
4:     if func = NULL then
5:         Finalize modification and terminate SLX
6:     end if
7:     if DB(func).coverage() ≥ THRESHOLD then
8:         strategy ← AGGRESSIVE_STRATEGY
9:     else
10:        strategy ← CONSERVATIVE_STRATEGY
11:    end if
12: else
13:    func ← func from A(H(func))
14:    strategy ← GetNextStrategy(A(H(func)))
15: end if
16:
17: //If no strategy for func, continue to the next function.
18: //Else, generate and assess a hypothesis for func using strategy
19: if strategy = NULL then
20:    HypothesisStrategySelection(DB, NULL)
21: else
22:    HypothesisGen&Assess(func, strategy)
23: end if
```

Fundamental to speculative analysis is selecting a strategy for generating the SLX hypothesis for each function. A hypothesis strategy consists of:

- analysis methods to produce the SLX hypothesis, and
- metrics for assessing the resulting hypothesis.

The remainder of speculative analysis performs the chosen strategy. Details of strategy selection in SLX are documented in Algorithm 1.

The database produced during analysis preprocessing, referred to as DB in Algorithm 1, is an input to every instance of strategy selection. In SSM, the hypotheses that are generated after strategy selection are assessed both in speculative analysis and speculative modification synthesis. For SLX, the result of these assessments are recorded in the hypothesis assessment summary, referred to as $A(H(func))$ (read as assessment of the hypothesis for function $func$). $A(H(func))$ is a second

input if strategy selection has to be reentered because the previous hypothesis was found to be unacceptable and has to be refined.

In SLX, the initial strategy selection for each function is based on the instruction coverage that the function achieved during execution of the test suite (recorded in DB). If instruction coverage is above a stakeholder-defined threshold, SLX will select an *aggressive* strategy to generate a hypothesis. We define an aggressive strategy as:

- attempting to recover data necessary to apply all three SLX modifications, and
- recovering the most variables in the stack frame.

A *conservative* strategy is selected if the coverage does not exceed the threshold. A conservative strategy uses analyses that recover fewer variables. In our implementation, the instruction coverage threshold is adjustable by the stakeholder. The justification for this approach to strategy selection is that, once a function modification is produced (in speculative modification synthesis), the modified SOUP will be executed to further assess/evaluate the hypothesis.

If strategy selection has to be repeated for a function, the assessment summary (i.e., $A(H(func))$) becomes the second input to that selection, indicating an invalid hypothesis was generated for the function. The summary is then used to aid in choosing the next strategy. In our implementation of SLX, a new strategy is selected only if $A(H(func))$ indicates

- one or more of the SLX modifications was assessed to be unacceptable, or
- the variable layout was considered unacceptable.

If any other component of the hypothesis is invalidated, no method for refining strategy selection is defined. Thus in that case, an unacceptable hypothesis will result in the function being excluded from further analysis and modification.

The metrics to assess hypotheses used by SLX are too numerous to enumerate here; however, most metrics involve examining the SOUP disassembly for instruction patterns. For example, some instruction patterns indicate a violation of assumptions upon which SLX relies to locate

canary values at run-time. In that case, adding canaries is an unacceptable modification. If one or more of the SLX modifications cannot be acceptably applied, the next strategy is to apply the remaining acceptable SLX modifications. If all three are invalidated, the function is excluded from modification. If the variable layout is unacceptable, the next strategy will use a different variable layout recovery approach.

Variable Layout Inference

All existing methods of variable layout recovery have benefits and weaknesses, and can produce erroneous and/or incomplete results [12, 80]. Many are not publicly available and difficult to engineer. Further, the stakeholders might be unwilling or unable to spend resources on developing these methods. For the purposes of this SSM case study, using the most sophisticated variable recovery approaches is not necessary nor even desirable for our use case. The important concept is how SSM works to balance risks of any selected variable recovery approaches. As such, SLX uses a simple layout recovery heuristic based on stack memory accesses (i.e., offsets into stack memory) seen in a function's disassembly.

For simplicity, SLX uses a total of four variable layout recovery inferences, three of which are characterized by the type of memory accesses:

1. All Offsets Inference (AOI)
2. Scaled Memory Offsets Inference (SOI)
3. Direct Memory Offsets Inference (DOI)

The fourth inference simply treats the entire stack frame as a single "variable", i.e., no internal structure of the stack frame is hypothesized. This fourth inference is referred to as the Entire Stack Inference (ESI). Figure 7.3 illustrates the results of these inferences for a sample disassembly.

SLX favors inferences that produce the most variables, and progressively backs off to inferences producing fewer variables as hypotheses are invalidated. If all layout inferences have been invalidated, the function is excluded from modification.

Function Disassembly	Source Code Semantics	Types of Memory Access
(0) push %ebp		
(1) mov %esp,%ebp		
(2) sub $0x38,%esp		# stack allocation
(3) movl $0x0,-0xc(%ebp)	; auth = 0	# direct offset
(4) …		
(5) mov %dl,-0x1c(%ebp,%eax,1)	; buf[i] = pwd[i]	# scaled offset
(6) cmpl $0x0,-0xc(%ebp)	; if (auth)	# direct offset
(7) …		
(8) leave		# stack deallocation
(9) ret		

Figure 7.3: Disassembly code fragment for the source code shown in Figure 7.1

7.5.3 Speculative Modification Synthesis

The hypothesis and the associated assessment summary generated by speculative analysis support strategy selection for modifying the function. The strategy specifies:

1. the manner in which the modification is produced, and

2. how the modification is assessed using a set of SLX-specific assessment metrics.

In the current instantiation of SLX, all modifications specified in the hypothesis are implemented using Software Dynamic Translation (SDT) [81]. SDT inserts a virtualization mechanism into the binary program, and this mechanism effects the changes necessary for the modifications during execution. Thus, the primary focus of modification strategy selection is to choose one or more assessment metrics.

Dynamic analysis is the primary mechanism used to determine assessment metrics. Synthesized modifications are executed using the test suite generated in preprocessing. If the behavior (i.e., the observed output of the program) deviates from the behavior of the unmodified program, the modification is invalidated.

SLX specifies three dynamic analysis assessment metrics, one of which is chosen by modification strategy selection:

- ***No assessment:*** If the function has no instruction coverage when executed, dynamic analysis provides no meaningful assessment. In this case, speculative analysis generates a conservative hypothesis that stakeholders accept without further validation through dynamic analysis.

- ***Execution of an error amplification:*** Canaries can help expose a variable layout error, but some functions cannot safely apply canaries. In that case, *error amplification* (see Section 3.2.3) is applied by randomizing the variables without padding and then testing, and repeating this process several times. If the variable layout hypothesis is incorrect, multiple randomizations might expose deviations in behavior.

- ***Execution of the candidate modification:*** If canaries can be applied, the modification is generated as specified in the hypothesis, and this modification is assessed.

If the modification is valid, SSM iteration will continue until all functions have been processed. If the modification is invalidated through assessment, SLX generates an assessment summary rejecting the hypothesis for the function. The assessment summary is sent back to speculative analysis to re-initiated SSM iteration in order to reconfigure the hypothesis. In our current SLX implementation, the variable layout is assumed invalid, and a less aggressive layout is chosen (i.e., layouts with fewer variables).

In this implementation, iteration within speculative modification synthesis, i.e., synthesis iteration, is part of a process optimization, further described in Section 7.6. Future implementation of SLX could enhance synthesis iteration, if, for example, overhead was a significant concern. Modification assessments that indicate excessive run-time overhead could result in re-configuring the modification (e.g., placing fewer canaries), or using only a subset of SLX modifications (e.g., removing canaries).

7.5.4 Termination of SSM Iteration and Finalization

Modification ends when a termination condition is reached (see Section 3.2.4). The implicit termination condition is when hypothesis generation and modification has been completed on all functions (lines 3-6 of Algorithm 1).

SLX is also implemented with an adjustable timeout termination condition, i.e., when the time to modify a given piece of SOUP elapses beyond a defined threshold (the default timeout value is

12 hours). For the target programs of the SLX use case, the timeout condition was never reached since modification never took more than a few hours. Timeout checks occur asynchronously during execution of the SLX process.

When the termination condition is reached, SLX initiates a modification finalization process in order to package all modified functions into a single program. Additionally, finalization runs any generated test suites against the entire modification to validate that the composition of modified functions is acceptable. If this final test of the modification fails, the entire modified program is rejected, i.e., considered unacceptable, and an error message is produced as an output.

7.6 Process Optimization

In Chapter 3, a brief discussion is given about the potential to optimize the SSM process: the general process is described linearly, but engineers could choose to perform any manner of optimizations, such as parallelizing or using cloud computing resources. Initially, SLX was implemented as a simple linear SSM instantiation; however, to improve the time required to produce a modification, an SSM optimization was applied. Instead of modifying each function sequentially, we altered/refined the SLX process to produce a hypothesis for numerous functions simultaneously. Modification synthesis then applies each hypothesis to the associated functions simultaneously, thereby generating a single candidate modification for the binary.

If the candidate modification fails assessment, synthesis iteration is initiated. The set of functions to modify is cut in half, and each half is modified and assessed independently. Synthesis iteration continues, in a binary-search-like manner until the hypothesis or hypotheses causing the assessment failure are found. The invalid hypotheses are sent back to speculative analysis for refinement, through SSM iteration.

7.7 The Assurance Case Acceptability Model

The SSM model as described in Chapter 3 prescribes the use of an assurance case. In this case study, however, the necessity of an assurance case was not immediately apparent and therefore not a part of the initial SSM model concept. As such, SLX was initially implemented without any consideration for explicit models of acceptability.

This case study revealed that, as decisions were accumulated affecting the acceptability of SLX for a specific stakeholder, unstructured arguments were insufficient. Explicit rationales are necessary to understand SLX processes and the effect these processes have on the security, correctness, efficiency, etc., of generated modifications. While we had these rationales in mind in the construction of the SLX implementation, the complexity inherent to SSM processes made it increasingly difficult to defend and justify SLX modifications as being adequately secure, correct, efficient, etc.

In principle, the SLX acceptability model is a complex collection of arguments and items of evidence to justify the acceptability of generated SOUP modifications. It includes acceptability criteria such as the instruction coverage threshold, risky instructions patterns, and configuration criteria such as the adequacy of the amount of padding used between variables. The form of acceptable inputs are also included, because an argument is need that these inputs exercise the program adequately and represent normal, i.e., non-malicious, execution adequately.

We then investigated the use of assurance cases as an acceptability model within the general SSM model to facilitate understanding of what the implementation provides the stakeholder. Examination of the SLX assurance case proved to be a complex investigation in its own right. As such, we performed a separate case study examining the utility and form of assurance cases in SLX and in SSM in the next chapter (Chapter 8).

7.8 Results

To assess SSM, SLX was evaluated with respect to produced modifications in terms of security efficacy, correctness, and efficiency. This assessment is performed on an early and simplified version of SLX. SLX has since been altered continuously to account for new operating context and stakeholder demands. We also provide some observations about the implementation, use, and maintenance/alteration of SLX to account changing demands.

We note that our results were obtained to assess SSM, not to assess our implementation of SLX. Those who might consider our implementation of SLX unacceptable can alter the implementation and acceptability model to fit their needs. The important concept is how SSM works with respect to a given use case. Data presented about SLX demonstrates the potential of SLX, the range of flexibility stakeholders have in implementing SLX, and how stakeholders might use SSM to effect trade-offs in order to balance risks to suit their needs.

7.8.1 Efficacy

Thwarting Attacks

The security efficacy of the three modification techniques used by SLX (variable padding, variable reordering, and canaries) have been studied extensively in the literature [72, 70, 71, 79, 69]; hence, part of the argument for the security efficacy of SLX is based on this prior work. To provide some empirical evidence of the potential security efficacy, we also applied SLX to a suite of sample vulnerable functions of relevance for our stakeholders, specifically the Wilander overflow suite [82].

The Wilander suite contains twelve stack-based buffer vulnerabilities. Table 7.1 provides the summary results of ten separate applications of SLX for each of the twelve attacks (classified by the targeted data). The result of Wilander attacks after applying SLX is classified as one of the following for each attack attempt:

Successful: The attack was unaffected by the modification, i.e., SLX provided no defense.

Attack Targeted Data	# Successful	# Detected	# Prevented	# Segfaulted
1) Local Function Pointer	0	10(100%)	0	0
2) Parameter Longjmp	0	10(100%)	0	0
3) Local Function Pointer	0	1(10%)	6(60%)	3(30%)
4) Parameter Longjmp	0	1(10%)	8(80%)	1(10%)
5) Return Address	0	10(100%)	0	0
6) Base Pointer	0	10(100%)	0	0
7) Local Function Pointer	0	10(100%)	0	0
8) Local Longjmp	0	4(40%)	6(60%)	0
9) Return Address	0	4(40%)	4(40%)	2(20%)
10) Base Pointer	10(100%)	0	0	0
11) Local Function Pointer	0	5(50%)	4(40%)	1(10%)
12) Local Longjmp	0	0	8(80%)	2(20%)

Table 7.1: Results of SLX applied to Wilander buffer overflow attacks classified by targeted data of attack

Detected: The canaries used by SLX detected a buffer overflow violation and terminated the program prior to the attack succeeding.

Prevented: Randomization of variables placed the target data of the attack out of the path of the attack. Hence, the form of the altered program no longer allows the attack to succeed in its original form.

Segfaulted: The attack crashed the program but did not succeed in achieving its purpose.

SLX was able to stop or detect the attack in all but one case. One attack succeeded because SLX was unable to infer variable boundaries with sufficient granularity for the vulnerable function (i.e., the hypothesis was sound but insufficiently precise).

Precision

While the security efficacy of the three modifications adopted by SLX can be a topic of debate, the SLX modifications can only provide a security benefit when they are applied to functions and they are most effective when the precision of detected variables is maximized. For example, in the above experiment with Wilander, one attack was able to succeed because the precision by which variables were detected was not sufficient.

To provide some indication of the potential efficacy of SLX on the targeted programs of our use case (Section 7.4), SLX was also applied to binary programs compiled with gcc and -O3

	Program Size in KB	Total Functions	Xformable Functions	Xformed Functions	Avg # of Variables Detected Per Function
chgrp	175	193	101	100	4.08
chmod	172	182	97	96	4.08
cp	271	281	154	153	4.49
dd	156	170	86	85	3.69
du	296	267	150	149	5.37
install	244	279	149	148	4.23
ln	163	174	94	93	3.96
mkdir	131	139	70	69	3.35
readlink	145	155	81	80	4.09
tail	149	156	85	84	4.10
touch	152	145	72	71	3.48
Average	**186.73**	**194.64**	**103.55**	**102.55**	**4.08**

Table 7.2: SLX statistics on CoreUtils

optimization for eleven of the Linux core utilities on Ubuntu 10.04 LTS, shown in Table 7.2. Only statically-linked functions were considered for transformation. We omit dynamically linked libraries both to simplify the experiment and since libraries should be transformed and evaluated separately. Libraries only need to be transformed once, after which they can be reused by any number of binaries. The suite of eleven Linux core-utility programs comes with a comprehensive set of test cases provided by the developers, and these were used for the validation step in SLX. Such a set of tests provides a more desirable situation than SLX is likely to encounter in practice. Nevertheless, the test suite is representative within the defined use case.

Table 7.2 illustrates both the characteristics of the modified programs in terms of size and functions, and the precision of variable detection. On average, 195 functions per program were found in the eleven sample programs. SLX determined that 53% of the functions were candidate transformable functions, meaning that SLX found patterns indicating that the function had local variables as well as stack allocation and deallocation patterns. Of the candidate functions, only one function, `term_proc`, could not be transformed (i.e., any modification of the function resulted in failing assessments). The failure to transform term_proc was due to the inability of our analyses to identify the stack deallocation point in the function.

AOI, the most aggressive variable layout inference, was used to transform 94% of the candidate functions successfully, with an average of four variables per stack frame. SOI was used on 4%, with an average of 1.8 variables found per stack frame. ESI was used on the remaining 2%. Previous work by Balakrishnan and Reps [12] suggest that using an AOI inference can successfully detect

variable boundaries 83% of the time[1]. The extensive use of AOI for these programs suggests that SLX is providing adequate precision most of the time.

The ability of SLX to use different variable layouts for each function illustrates the use of SSM process principles. Functions that cannot or should not be transformed are filtered through the preprocessing phase. Additionally, as modifications are generated, SLX refines hypotheses about how the program should be modified. While in some cases this resulted in a decrease in precision, the refinement was able to balance concerns of correctness.

7.8.2 Correctness

For a modification to be correct it should preserve the *intended* (not necessarily original) program semantics (some modifications are made with the explicit intent of modifying program semantics; hence, these semantics alterations are "*intended*"). This implies modifications should be made:

1. precisely, i.e., all components that need to be modified are modified and no others, and

2. accurately, i.e., what is modified is modified in an appropriate manner.

Modification correctness is in direct conflict with the security efficacy. Attempting to maximize the number of functions modified, the number of modifications performed per function, and the number of variables detected will increase the ability for SLX protections to provide a security benefit, but it also increases the risk that the modified program will no longer behave as intended.

Observations of Correctness Complexity

Correctness in the context of SLX corresponds to what instructions should be modified, how, and the variable boundaries detected. We know of no method at present to fully represent the correctness of SLX modifications with respect to these three characteristics.

[1] At present, we know of no study where variable recovery mechanisms in binary programs are compared with what the compiler actually intended. Most approaches rely on debugging information which corresponds to source code, and can be misleading. Therefore, the measurement of 83% variable recovery cannot be taken as an absolute fact, but a rough figure.

To illustrate the complexity of determining the correctness of SLX modifications, consider variable layout in isolation. Even if the variable layout can be determined to be 100% correct, correctness of the modification that uses the layout is not guaranteed. The modification itself might break expected program behavior. For example, adding padding between variables might result in a stack overflow.

Even if variable layout alone was the single issue affecting correctness, at present no analyses exist that provide 100% precise and sound recovery of stack frame layouts in a binary program. Most methods rely on debugging information produced by a compiler to determine layout accuracy (see for example, see previous work by Balakrishnan and Reps [12]). This notion of accuracy is with respect to the source code and not to what the compiler actually produced. A compiler can alter the sizes and number of variables, as well as add empty space and temporary variables not in the source code. Basing variable layout correctness on this concept can therefore be misleading.

Finally, comparing layouts against compiler generated layouts can be problematic since correctness of variable layout can be subjective. For example, a single buffer in the source program might be used to encoded a multi-dimensional array. Is it correct to infer this data structure is one variable, or multiple variables? Similarly, a developer might declare a struct within a function. Is it correct to consider the struct as one variable, or can each field within the struct be considered an independent variable? Answers to these questions depend on how the software is used and the nature of the modification.

Justifying Correctness

Because of the complexity of determining correctness, the initial implementation of SLX established confidence in correctness primarily by the ability for modified software to pass a series of regression tests. All programs modified with SLX for this study continued to operate after modification when tested against a given test suite; hence, the modification is believed to be correct.

Subsequent refinements of the SLX implementation added more assessments to verify inputs meet a stakeholder-defined standard, e.g., a threshold of instruction coverage was defined. Further,

SLX was altered to check for patterns in the binary program and in the structure of functions that are known or are believed to be problematic, i.e., cannot be successfully modified without breaking expected program behavior. As subsequent versions of SLX added more assessments for correctness, additional methods for addressing assessment failure were also added. For example, in addition to changing the variable layout inference, the configuration of modifications can also change (e.g., how much padding to use), or the combination of modifications that are applied can vary (e.g., canaries might risk breaking the program for some functions and therefore not applied in these cases). With each additional kind of variation added to SLX, the complexity in what SLX can produce increased, as well as the complexity for the rationale for believing the modification is correct.

The primary result from attempting to evaluate correctness for SLX is that SLX (and therefore SSM) must be supported by a more structured and explicit acceptability model/rationale. Because of the interactions and complexities of correctness concerns, this information led to the motivation to include an assurance case as part of the SSM model to organize and simplify the rationale for correctness.

7.8.3 Efficiency

The efficiency of SLX can be examined from two points of view:

1. **Developmental Efficiency**: The resource consumption SLX uses to generate modified SOUP.
2. **Operation Efficiency**: The resource consumption of the generated SOUP modifications.

These two concepts of efficiency are discussed in the following subsections.

Developmental Efficiency

In the initial implementation, SLX modified functions one at a time; hence the processing time required is linear with respect to the number of functions to modify. The primary factor in processing time for each function is the time required to test the modified function. Our latest implementation

	% Overhead of All SLX Mods	% Overhead without Canaries	Program Size in KB	Total Functions	Xformable Functions	Xformed Functions	Avg # of Variables Detected Per Function
bzip2	49.66	4.70	66	101	63	63	5.33
lbm	6.08	2.50	17	45	20	20	6.30
mcf	119.29	2.40	17	49	23	23	4.13
milc	10.32	1.69	125	281	158	158	6.72
sjeng	67.50	38.00	156	176	97	97	11.93
libquantum	9.29	3.91	45	136	77	77	6.40
Average	**43.69**	**8.87**	**71.00**	**131.33**	**73**	**73**	**6.80**

Table 7.3: SPEC 2006 overhead results

uses a binary-search-like optimization to allow multiple function modifications to be transformed and tested simultaneously (Section 7.6). We have observed many cases where this approach is more efficient than modifying and testing one function at a time, but, as the number of invalid hypotheses increases, processing time can become greater than assessing one function modification at a time.

For our use case (Linux coreutils), modifications took on the order of a few minutes to a few hours. We considered this level of resource usage to be acceptable, because testing modified functions is conducted before deployment and is only done once (provided the modification is determined to be acceptable). If SLX were extended to modify programs with thousands of functions and test suites requiring minutes to process, modification development time would probably exceed acceptable limits with the current implementation (e.g., modifying a program takes several days, weeks, or longer). In this case, other optimizations can be applied to the SLX process. For example, instead of modifying each function one at a time, or in a batch, all functions could be modified and assessed in parallel (e.g., using cloud resources, one computer for each function), and/or modifications can be tested with a subset of the test suite that actually executes the modified function.

Operational Efficiency

The operational efficiency depends largely on the associated stakeholder demands. For example, if stakeholders require a claim that the residual risk of a successfully attack be very low, then the modification will have to include more countermeasure thereby increasing the computational resources required (e.g., software run-time).

The principles of SSM allow stakeholders to assess modifications based on any competing constraint, including efficiency. For the initial SLX prototype, efficiency was not considered in the assessment of modifications; however, to illustrate the potential range stakeholders can select from, SLX was applied to a set of SPEC 2006 benchmarks [83] of comparable size to the target software of our use case (6 programs in total) to evaluate potential run-time overhead. The statistics about the characteristics of these programs and the overhead results are summarized in Table 7.3.

To illustrate the range of efficiency trade-offs, SLX was applied with and without canaries. The run-time overhead incurred by the modified programs ranged from approximately 9% without canaries to approximately 44% with all SLX modifications including canaries. The reported overhead of the software dynamic translator used by SLX is 7%. Acceptability of overhead is for the stakeholders to decide, but, given the protection that SLX offers, these levels of overhead seem reasonable.

These overhead results also indicate the flexibility stakeholders have. By limiting how canaries are placed and checked or how much padding is used, stakeholders can expect to achieve somewhere between 7% and 44% overhead. Stakeholders can expect to see higher overheads if canaries are placed and checked more frequently or if variables are not aligned on cache boundaries. Stakeholders can alter their acceptability model to find an appropriate balance between benefits and efficiency to fit their needs, or attempt inefficient modifications, and back-off to more efficient modifications through synthesis iteration (see Chapter 3).

7.8.4 Observations

The development, use, and maintenance of SLX led to several general observations about SLX and SSM. Some of the most important observations are described in the following subsections.

Assurance Models

The most profound result from this case study is the need for an explicit model of SSM assurance. Initially, the SSM model did not include the concept of an assurance case, or any explicit model

of assurance. As more and more assessment and strategy selection criteria were added to SLX, an explicit and structured approach became increasingly necessary.

Further, while we can provide some evidence that SLX provides some utility (as presented throughout Section 7.8), without a structured assurance argument, it is not clear what the evidence actually supports. An assurance case provides a succinct and rigorous justification of the acceptability of SSM modifications. For SSM to be practical, an assurance case is fundamentally necessary (further exploration of the assurance case is given in Chapter 8).

Expert Opinion

In developing SLX, we observed that SSM modifications might require substantial technical expertise and insight. For example, the SLX modifications require a detailed understanding of:

1. how memory is managed on target machines, and

2. how target compilers (gcc) can optimize and alter stack memory.

The inclusion of assurance cases into the model also necessitates assurance case experts to develop and structure a rigorous argument that refers to complex evidence derived from the operation of the SSM process.

The mechanics of the SSM model are also fundamentally tied to a detailed understanding of the problem domain. Experts are required for determining suitable hypotheses, and deciding how to assess and refine them. A primary challenge in developing SLX was finding metrics to assess hypotheses, and in developing SLX these metrics were developed by trial and error. A test coverage metric was needed, for example, when running tests to determine whether the modification had affected program semantics. Test coverage metrics for binary programs are few, and no thresholds are generally accepted. SLX development required several judgments such as these, highlighting the importance/need for a structured assurance case to organize these judgments, and to expose these judgments for further assessment.

Maintenance

SSM provides no guarantees that a final modification will not fail at some point in the future. Programs modified by an SSM process might be left in a state where necessary/expected semantics have been changed, or the desired dependability property might be absent with no indication. We have observed SLX modifications fail post development either by allowing a buffer overflow attack to succeed or by causing the executing program to crash. In these cases, the cause of the failure is reviewed, and the SSM process corrected.

In our observed use of SLX over time, the need to adjust the process is essential to the practicality of SSM. For example, SLX modifications are partially based on assumptions about how the compiler generates executables, i.e, compiler idioms. As new compilers are released, these idioms change and require maintenance of the SLX process. Additionally, SLX was used for other projects, requiring continuous alterations to adjust concepts of acceptable modification.

While not present in the SSM process model, SSM can be thought of as a component in a larger modification engineering life cycle. In essence, SSM can have a post deployment *maintenance* component, to correct/reconfigure an implementation and reapply it to software when failures are observed.

Essentially, the SSM process is never finished, but instead evolves constantly to adjust to new programs and new demands by stakeholders. The need for continued maintenance further supports the need for the SSM assurance case. As we attempted to alter the implementation of SLX to allow for more variability in how modifications are applied, determining what to alter, how to alter it, and what effects the alteration might have on other components of the process became intractable. If an assurance case were generated for SLX, we could focus on applying the modification in the argument first to determine the proposed change improves the quality of modified software before implementing the solution.

7.9 Results Summary

The goal of this case study was to answer the questions presented in Section 7.1. The answers to these questions are summarized as follows:

Answer to Question 1: SSM is Feasible

SLX was implemented as an instantiation of the SSM process to protect against a real-world class of vulnerabilities within a very specific operating context. By applying SSM to the circumstances of the case study demonstrates the potential feasibility of the concept.

As stakeholders define more alternative modification techniques and configuration and assessment techniques, the process becomes increasingly difficult to engineer requiring more development resources (i.e., increased SSM development risk) and the resources required to produce software modifications also increases (i.e., increased modification development risk). If engineers carefully consider SSM concepts early and often in development, we anticipate that SSM development risks can be mitigated through careful organization and documentation of the SSM implementation. Modification development risk can be handled by either one or both of the following options:

1. The SSM implementation considers the possibility of modification development exceeding acceptable thresholds, i.e., the process will assess modification development risks, and choose alternatives that are more feasible.

2. The SSM implementation can take advantage of optimization techniques, such as parallel and cloud computing, to attempt a large number of modification alternatives and perform associated assessments simultaneously.

We therefore conclude that the SSM process appears feasible. Generalizing this conclusion is not possible from a single instance, but we do not anticipate any other limitations affecting SSM feasibility.

Answer to Question 2: SSM is Practical

There are two concerns about the practicality of the SSM process:

- First, can SSM be used to generate modifications of practical value, i.e., modifications that achieve some technical goal within defined constraints? SLX, as a implementation of SSM, was shown to produce software modifications with some security value. Previous publications on the modifications SLX uses also indicate the potential for the modifications applied by SLX. Of key importance, however, was the ability for the SLX implementation to directly and explicitly account for concerns typically overlooked in the literature. Specifically, assessment and refinement activities, as specified in SSM, allow SLX to balance constraints and adjust modifications to meet the specific needs of the system stakeholders. This concept in particular is necessary to make SOUP modifications practical, regardless of the security benefits any modification provides. The use of the SSM process in this study allowed us to balances constraints as necessary, and therefore, allowed for practical application of SLX modifications.

- The second concern affecting the practicality of the SSM process is the difficulty of implementing and maintaining an SSM process. SLX, while developed to defend against a very specific class of attacks within a very specific operating context, presented unexpected complexity in balancing security with other constraints. The initial version of SLX intentionally minimized this complexity by narrowing the operating context and needs of the stakeholders. Over time, these needs have been altered necessitating additional assessment and refinement loops, increasing the complexity of the implementation, the interactions between decisions, and the space of acceptable software modifications. Using an SSM process, we were able to accommodate alternative operating contexts. Although the complexity of SLX developed to a level that was increasingly difficult to maintain and understand, we did not develop SLX from a finalized concept of SSM. Not developing with SSM concepts early on can therefore limit the practicality of the SSM approach; hence, engineers should start developing with SSM concepts early (discussed further in Chapter 9).

The results of this case study suggest that SSM allows stakeholders to configure modifications to meet there needs and manage risks as they see fit, which makes the SSM process practical. We observe that the practicality of the process is primarily limited by the stakeholder's ability to understand what the process is providing (i.e., understanding their choices, configurations, etc.), and highlights the need for structured, rigorous, and explicit SSM assurance cases.

Generalizing this result is not possible from a single instance of the SSM process, but we observe that the kinds of security problems addressed by SSM, i.e., how and when to apply software modifications, are likely applicable for other operating contexts (i.e., other stakeholders, organizations, kinds of programs, etc.) and security concerns. Further exploration of the general applicability/practicality of SSM for a different security modification is described in Chapter 9.

Answer to Question 3: The SSM Model Requires Assurance Cases

The SSM process model was continually refined during the implementation and assessment of SLX; hence, process mechanics of SLX are completely captured by the SSM process model. To avoid over fitting the model to SLX, the model was also developed and scrutinized analytically and then assessed using SLX. This study suggest the process component of SSM captures the necessary activities of speculative modifications.

The model, however, in the form discussed in this case study has been shown to be incomplete. The model fails to represent SSM modifications in a form that permits adequate comprehension. Additionally, it did not provide mechanisms to reconfigure decision processes in a clear way.

A fundamental result of this study was that the SSM process is used to justify the acceptability of a SOUP modification; however, the process by itself does not provide any explicit justification. In practice, an explicit justification is not only useful but would likely be required.

The solution we have adopted in the SSM model is to document the rationale of SSM decisions in the form of an assurance case. An examination of the utility of assurance cases applied to SSM is discussed in the next chapter (Chapter 8).

Chapter 8

Case Study: Exploring the SSM Assurance Case

The previous chapter (Chapter 7) presented an exploratory case study to illustrate and evaluate the process mechanics of SSM. A major result of that study was the need for more explicit assurance models to justify the acceptability of SSM modifications to make the SSM model practical. This chapter presents an extension exploratory case study of Chapter 7, targeting the same software modification for evaluation, SLX; however, the focus is to address the research question:

How can engineers rationalize and understand the acceptability of modifications an instantiation of SSM produces?

This case study evaluates the form and utility of assurance cases applied to SSM by constructing an assurance case for SLX. For a review of the motivation and description of SLX, refer to Chapter 7.

8.1 Case Study Goals, Overview and Scope

The goal of this study is to determine if assurance cases can be used as an acceptability model for the SSM process. To understand the form and utility of assurance cases for SSM, assurance cases are applied to SLX to answer the following research questions:

Question 1 Can assurance cases be feasibly applied with the SSM model?

- Can assurance case technologies be extended for SSM?
- Can an SSM assurance case be constructed?

Question 2 Does the SSM assurance case enable the practical application of SSM?

- Does the SSM assurance case facilitate understanding the acceptability of SSM modifications?

The motivation of applying assurance cases to SSM is to provide a single argument that will justify why any given modification produced by SSM is acceptable (this rationale is further discussed in Chapter 3). The form and mechanics of how the assurance case would meet this requirement was part of the experiment of this case study.

Initially, the SSM assurance case concept consisted only of a fitness argument. The fitness argument structure defined in Chapter 4 was developed and applied as part of this case study. The initial goal was to generate a traditional assurance case for SLX based on the a fitness argument structure. The construction of the argument led to the discovery of the fundamental limitations in traditional argumentation methods, e.g., the inability for the argument to capture to variability of SSM modifications. *The addition of a success argument and selection argumentation to the SSM assurance case is a key result of this case study*.

The scope of this study was to develop an assurance case for SLX to a point where the goals of this study were met (i.e., where the above research questions are be answered), not to develop a complete assurance case. Since no assurance case was developed prior to implementing SLX, full recovery of the assurance case would require both reverse engineering out of the SLX source code the implied arguments, and fully developing those arguments that were not fully rationalized. This is beyond the scope of this research. Concerns of confidence (described in Chapter 5) are also omitted to simplify the study.

8.2 Applying the Security Fitness Argument

This section provides an overview of the application of the fit-for-use argument structure for security from Chapter 4 to SLX. The general structure consists of four levels of argument, and also involves an iterative argument repair mechanism as "weaknesses" in the argument are discovered. A weakness in this context refers to a lack of specificity in the argument resulting in an inadequately supported claim. This section discusses the construction of the SLX argument using the principles of these four levels and the iterative repair mechanism.

8.2.1 Argument Level 1 - Fit For Use

The top-level argument structure is illustrated in Figure 4.2 of Chapter 4. The instantiation of this pattern for SLX is shown in Figure 8.1. The pattern is instantiated by identifying the following (further described in subsequent subsections):

- The operating context.
- The threat model in which mitigation techniques will be applied.
- The assets to protect.
- The security properties of the identified assets that the software modification should provide.
- Any additional constraints on the modification.

Operating Context

The operating context describes any information about the environment in which the SLX modifications will execute as well as the general characteristics of the software to be modified (i.e., characteristics of the SOUP). The operating context of SLX (referenced within Context 1.2 as D1) is summarized as follows:

- SLX was developed to modify SOUP in raw binary form, compiled to machine code.

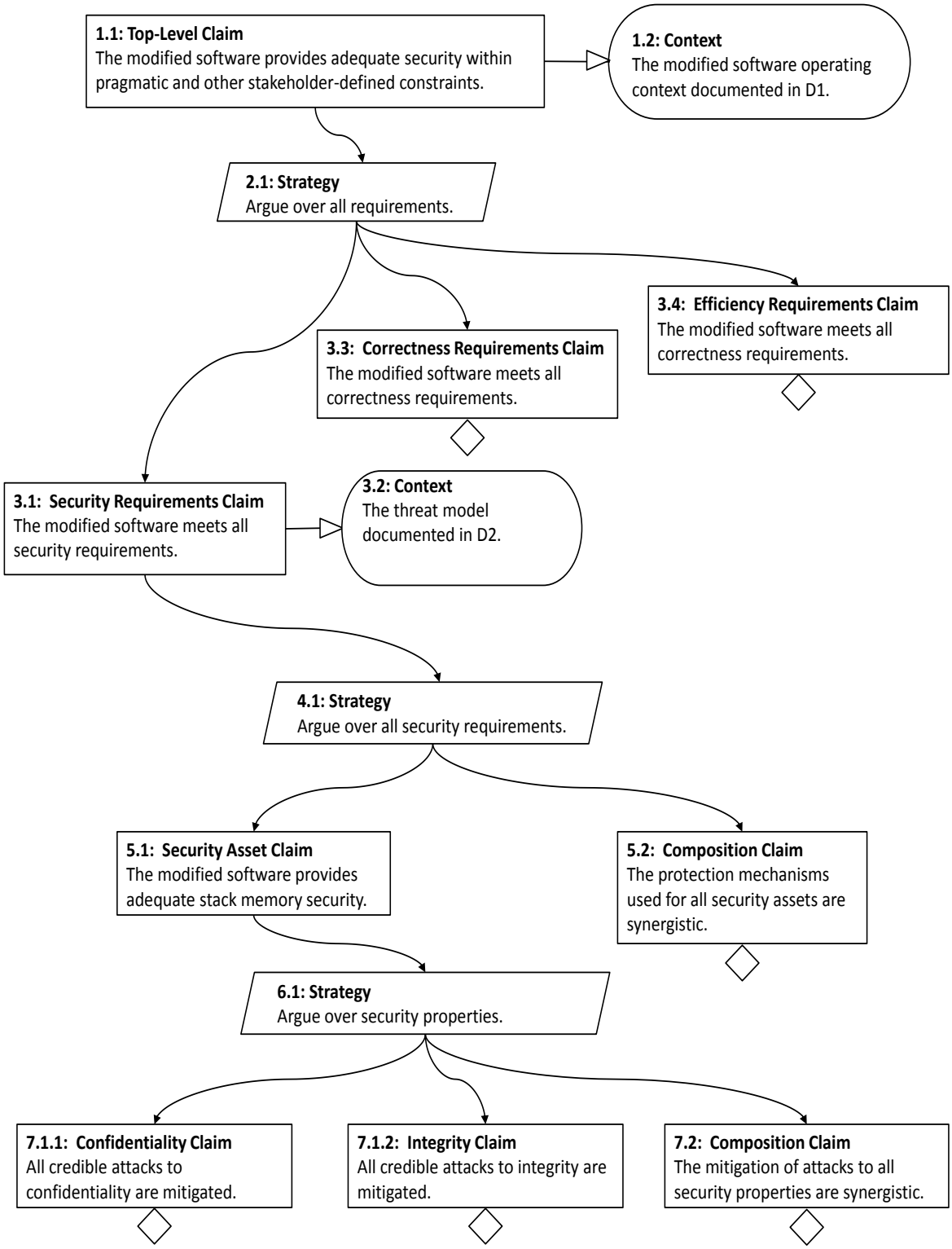


Figure 8.1: Top-level SLX fitness argument structure

- Development information, especially program source code, and debugging information are presumed not available.
- SLX is intended for SOUP compiled using common compilers for common architectures and operating systems. In particular, SLX targets SOUP compiled by the GNU C Compiler (gcc) as well as the 32-bit x86 architecture and Linux operating system.

Threat Model

The threat model captures all assumptions about the system upon which mitigation techniques will rely. For SLX these assumptions (referenced within Context 3.2 as document D2) are:

- The adversary might have access to the unmodified SOUP and the program source code.
- The adversary can provide malicious inputs to the SOUP.
- The adversary does not have direct access to the hardware or the modifications SLX produces.
- The SOUP is free of self-modifying code and malicious software (e.g., intentionally planted backdoors and trojans).

Security Asset Identification

A security asset is a component of the software for which there is a credible threat of attack. Identifying appropriate assets was a subtly difficult challenge. For the purposes of this case study, SLX is used within the context of a hypothetical use case described in Chapter 7. The challenge in identify assets is the pull between demonstrating the benefits of SLX and demonstrating the practical application of SLX. These two goals are in opposition in that:

- If assets are defined out of the scope of SLX, then we will quickly find SLX has no benefit at all.
- Narrowing the assets too much would cherry-pick a scenario where SLX would be most beneficial, and therefore give a contrived and false impression of security.

SLX was primarily envisioned as a defense mechanism against buffer overflows that occur on the stack. This description is characteristic of a type of vulnerability, and not an asset. Abstracting the intent of SLX further, SLX can be described as a mechanism to protect the execution stack memory of a program. We could identify arbitrary assets within the stack for illustration; however, SLX is meant to modify arbitrary programs. We therefore decided to consider the entire execution stack the asset to protect (shown in Goal 5.1).

Security Property Identification

Relevant security properties depend on the needs of the stakeholders and the identified software assets. For example, had mission critical data been chosen as an asset to protect, the security properties of interest appear biased towards integrity and availability. Similarly, if sensitive or private data had been identified, such as credit card numbers, confidentiality appears to be of most importance. In this case study, a general concept of a security asset is identified, i.e., the execution stack, which can contain various kinds of data; hence, we chose both confidentiality and integrity (Goals 7.1.1 and 7.1.2) as the targeted security properties.

Additional Constraints

There are various assessments performed throughout the SLX process designed to validate that the modification will provide desired functionality (i.e., that the modification is correct). These assessments include:

1. assessment of the behavior of the executing program, and

2. assessments that examine the form of the SOUP.

For this case study, we included a goal for correctness (Goal 3.3) but left it undeveloped until we further examined how to argue security.

Efficiency is another constraints of practical concern; however, SLX was not designed to meet any specific efficiency requirement. Evidence presented in the previous case study suggests there

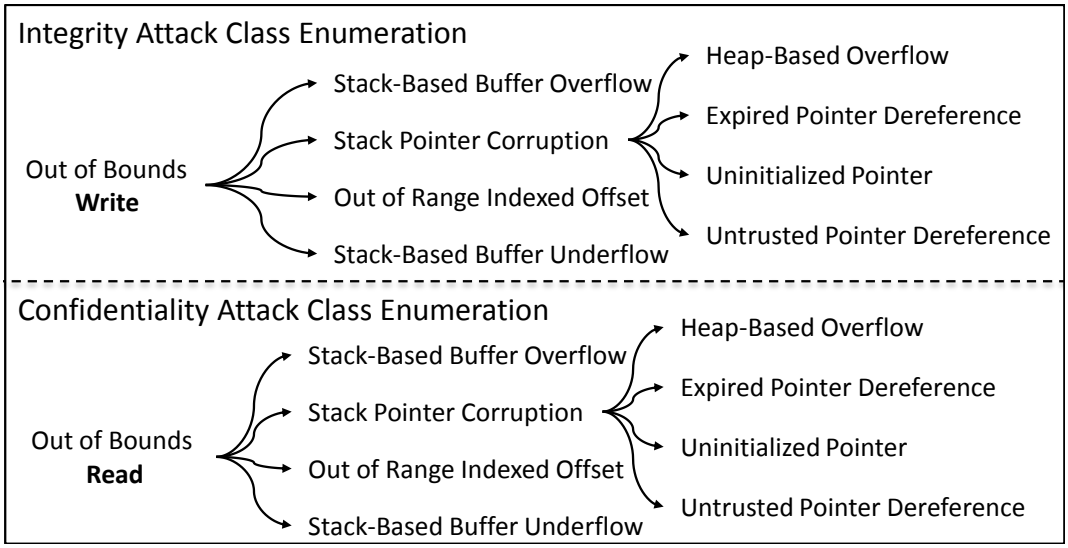


Figure 8.2: Stack memory attack classes

is a range of potential run-time overhead of SLX modifications, but, without a specific efficiency goal, the evidence does not provide any useful support. An efficiency goal, Goal 3.4, is included but left undeveloped until stakeholders define precisely what the limits of acceptable overhead are. If efficiency becomes a pressing concern, stakeholders can alter the SLX process to balance efficiency in the same manner in which correctness is balanced, i.e., through assessment and refinement.

8.2.2 Argument Level 2 - Attack Classes

For each security property, stakeholders must define the credible threats, i.e., the credible attack classes, that must be mitigated. An attack-class enumeration was constructed by consulting the Mitre Common Weakness Enumeration (CWE) [42]. Based on our judgment of credible attacks, a subset of attacks was selected from CWE-633 (Weaknesses that Affect Memory). These attacks were refined into two general classes, out-of-bounds write (attacks against stack integrity), and out-of-bounds reads (attacks against stack confidentiality). Further subdivision of attack classes under both of these general categories were identical. For simplicity, the attack class enumeration hierarchy is shown in Figure 8.2 out of context of the argument structure.

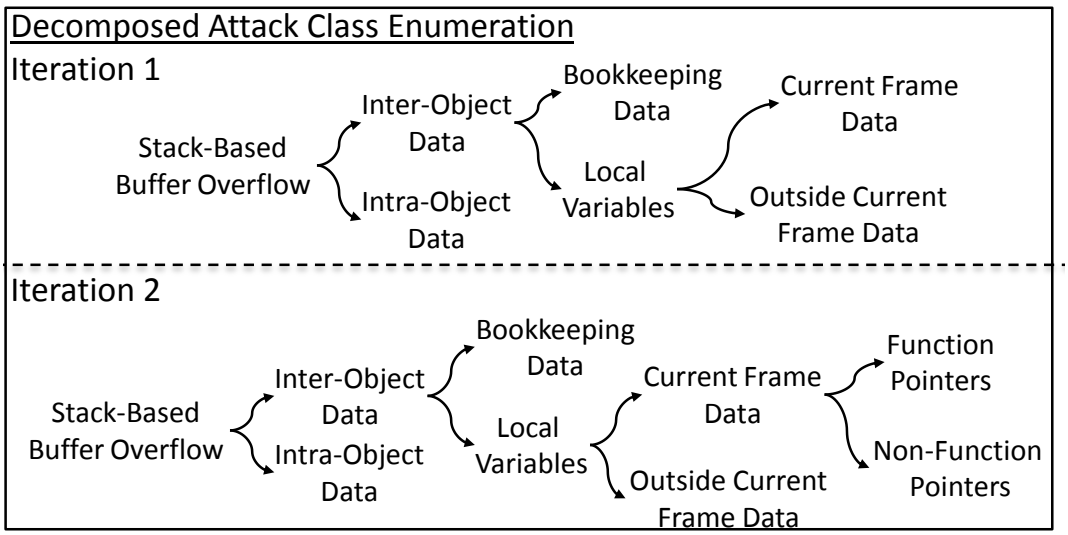


Figure 8.3: Example of two iterations of the decomposed attack class enumeration process on the attack class taxonomy shown in Figure 4.5. The first iteration subdivides the stack-based buffer overflow attack class. The second iteration further subdivides the "current frame data" decomposed attack class.

8.2.3 Argument Levels 3 and 4 - Iterative Argument Development

Refinement of the argument could not continue in a sequential decomposition from this point because of the interplay between the argument and analysis of the security protection mechanisms. The argument construction continues as an iterative process in which argument refinement drives analysis (see Figure 4.5 from Chapter 4). The initial attack-class enumeration was further subdivided as mitigation methods were assessed and mitigation arguments were constructed. In essence, the argument drove selection and evaluation of low-level security techniques to mitigate attacks in a process similar to Assurance Based Development (ABD) [23] (ABD is described further in Chapter 10). Subsequent analysis then suggested ways in which the argument should be further expanded (i.e., new decomposed attack classes were added).

In the first iteration of this process for SLX, assessment exposed several limitations requiring further subdivisions in the argument. For example, in SLX:

- There is no protection against buffer overflows within data objects (i.e., C structs).

- Local variables are protected by a different degree of protection from other bookkeeping data structures on the stack (e.g., the frame pointer and return address).
- Data within the currently executing stack frame is protected differently than data outside the frame.

The top of Figure 8.3 (iteration 1) illustrates how these issues were used to further subdivide the attack classification hierarchy for "stack-based buffer overflow attacks" affecting stack integrity (see Figure 8.2). For SLX, most decomposed attack classes were used indicate a lack of specificity in the originally identified software asset because of fundamental weaknesses specific to SLX.

For a given attack class, the general mitigation argument approach was to:

- expose the characteristics that SLX provides which might support an attack class mitigation claim,
- organize modification approaches into these mitigation technique categories, and
- recover the claim (or claims) that can be strongly argued for a given mitigation technique.

For example, consider developing a mitigation argument for the stack-based buffer overflow attack class that targets data within the currently executing stack frame (the specific enumeration leading to this attack class is shown in iteration 1 of Figure 8.3). In SLX, the use of canaries was the first mitigation approach examined. Canaries can potentially detect attacks for this attack class; hence, we attempted to recover a strong argument about what level of attack detection canaries can provide.

In the original version of SLX, canaries are checked at function call and return. This sparse placement of canary checks was justified to keep run-time overhead acceptable. Analyzing this decision to produce a convincing argument reveals that the efficacy of canaries depends on the type of data targeted by the attacker. Specifically, function pointers are protected more convincingly than other forms of data.

Attacks in which function pointers are corrupted with a malicious address are assumed to succeed only if used in a subroutine call. Since canaries are checked prior to call instructions, a

claim can be made that all function pointer attacks will fail for this particular attack class (provided canaries are completely and correctly applied).

For other non-function pointer data, the attack might be detected prior to the attack succeeding but detection depends on the frequency of canary checks. The frequency of canary detection is not set within the current SLX implementation. The difference between security claims that can be made between these two classifications of data results in the addition of two decomposed attack class, shown at the bottom of Figure 8.3. The revision of the "current frame data" decomposed attack class is further subdivided into attacks targeting function pointers and non-function pointers.

8.3 Results of Argument Benefits

Attempting to construct the security argument in the manner described above revealed a number of weaknesses in SLX [58], a significant result of the use of an assurance argument for SSM. The weaknesses cover a variety of topics, and their composition together with the details of the remainder of the argument show the substantial value of applying rigorous argument to software security enhancements, and by extension SSM. A summary of exposed weaknesses is provided in Table 8.1. Below, we provide a description of these revealed weaknesses, and describe the ways in which argument aided in their discovery.

8.3.1 Essential Trade-offs

Security, like many system properties, is not isolated. Frequently, trade-offs have to be made in which security might actually have to be weakened in order to make it practical. Exposing the need for trade-offs and what they imply is facilitated greatly by the security argument. An important example revealed in SLX is the use of canaries.

Canaries can incur substantial run-time overhead as a result of added functionality to set up and check canary values. To curb run-time overhead, the original implementation of SLX only checks canary values at function return and call instructions. As noted above, the structured reasoning

Result	Summary	Section Containing Further Details
Argument development exposes modification trade-offs.	Discovery that argument development aids in exposing the trade-offs within the configuration of a modification between security, correctness, efficiency, etc.	Section 8.3.1
Argument development exposes weakly mitigated attack classes.	Discovery that development of the argument aids in identifying mitigation techniques that are not well-suited for mitigating attack classes.	Section 8.3.2
Argument development exposes reliance on qualitative evidence.	Discovery that development of the argument aids in highlighting where the argument is based on qualitative evidence, such as expert judgment and intuition, which might be considered unacceptable.	Section 8.3.3
Argument development exposes alternative modification configurations.	Discovery that development of the argument raises doubts about the configuration of analyses and mitigation techniques and further aids in eliciting alternative configurations.	Section 8.3.4
Argument development exposes unhandled attack classes.	Discovery that development of the argument forces stakeholders to consider all credible attack classes, including those that might have otherwise been overlooked.	Section 8.3.5
Argument development exposes unanticipated benefits.	Discovery that development of the argument forces reasoning about the benefits of mitigation techniques exposing benefits that might not have been originally considered or reported.	Section 8.3.6

Table 8.1: Summary of results of assurance case benefits

provided by the argument exposed the implicit impact of this trade-off. Specifically, function pointers are strongly protected, whereas other kinds of data are not.

Exposing trade-offs enables analysis and explicit consideration of their (potentially unintended) effects. In this example, a solution that better protects non-function pointer data might be sought. One approach is to check canaries after a set number of instructions. Including regular canary checks allows a stronger arguments to be constructed at the cost of having to execute the additional checks.

8.3.2 Weak Mitigation

Attempting to argue a security claim frequently reveals weaknesses in the underlying technique that might not have been considered. An example in SLX is the use of padding between variables. Previous work using padding as a security enhancement provides some quantification of the security efficacy [72], but only for a very limited number of attack classes. Examining padding within the argument structure exposes this limitation, because producing a convincing, general and quantifiable argument for padding for many attack classes is not possible.

Confronted with a potentially weak modification might suggest that the approach should be abandoned; however, to do so might ignore other benefits. Instead, an argument can be constructed to show exactly to what extent and under what circumstances a particular technique should be applied.

Intuitively, using padding might increase the number of attack attempts required to succeed, depending on the nature of the attack class. If this increase in attacker effort either defeats the attack or can be shown to make attacks highly detectable (perhaps combined with canaries), a compelling argument might be made in favor of padding. The key benefit of using argument in this case within SLX was to highlight a potentially weak mitigation approach and to aid in exposing what the approach can fully provide, before abandoning it as a protection mechanism.

8.3.3 Qualitative Evidence

Quantifiable performance measurements frequently offer strong evidence to support an argument claim. In some cases, however, qualitative evidence is all that is available. Decisions concerning the adequacy of qualitative evidence are left to experts and ultimately to the system stakeholders. Before judgment can be applied, stakeholders must be made aware of what claims are affected and how. Without a comprehensive argument, the effect of qualitative evidence cannot be easily assessed. By applying security arguments to software security enhancements, details of the claims supported by qualitative evidence are revealed and can be reviewed.

An example in SLX is the adequacy of variable randomization, i.e., the reordering of variables on the stack as a mitigation technique. Randomization forces the attacker to perform a state-space search in order to find a variable of interest. Requiring a search is a useful deterrent in its own right, but a sophisticated attacker might be able to launch an attack that includes a search. In that case, the success of the search depends upon the entropy introduced by the randomization, and that can be quantified statistically.

Nevertheless, a significant benefit of variable randomization derives from the kind of data being randomized. If a variable is placed in the path of a buffer overflow, an attacker might be forced to corrupt that variable to effect an attack. If the program relies on a specific value of the variable, then the attacker will have to recover this value or face program termination.

Intuitively, there is a potential increase in attack effort, but exactly how often this occurs and to what extent such attacks are typically thwarted has no quantification. A decision is required as to whether this intuition is sufficient, or if additional research is needed to generate quantitative evidence.

8.3.4 Alternative Configurations

As arguments are constructed for an attack class mitigation claim, the process reveals characteristics of the mitigation techniques that could be altered to potentially improve belief in mitigation claims. Configuration options discovered as a result of this case study include:

- Where canaries are placed
- The size of canaries
- How often canaries are checked
- Remediation policies when attacks are detected
- The content of padded memory regions
- The size of the padding

- The algorithm used for variable randomization

Rather than claiming any particular configuration is weak or determining optimal configurations, we emphasize the benefit of argument to aid in exposing these configuration decisions.

8.3.5 Unhandled Attack Classes

One of the primary benefits of the argument structure and iterative attack class refinement process is the ability to reveal credible attack classes for which a given set of modifications are not well suited or possibly completely ineffective. The argument is essentially a comprehensive view of precisely what security claims can be made and where additional support is needed. Stakeholders can then use the argument to provide structured guidance as to how go about fully protecting the software.

An example of this argument benefit arose in the analysis of SLX as a defense against attacks on confidentiality. While SLX does provides some data diversity in the stack frame structure that could make confidentiality attacks more difficult, the evidence was found not to be compelling.

Additionally, canaries provide absolutely no protection against attacks on confidentiality. In examining the claim about the composition of mitigation methods for integrity and confidentiality (i.e., arguing the chosen methods do not conflict), a serious weakness with canaries was discovered. Since the canary is in a predictable location, attackers could use a confidentiality attack to recovery the canary, potentially allowing the attacker to overwrite the its value (i.e., bypass the canary) without detection.

Finding that SLX canaries can be subverted by a combination of confidentiality and integrity attacks suggests the possibility of reconfiguring canaries, such as encrypting canary values, or placing canaries in unpredictable locations.

8.3.6 Unanticipated Benefits

Applying argument to security enhancements can reveal unanticipated benefits that a modification provides. These benefits would not necessarily be revealed by ad hoc (as opposed to argument

based) analysis. The failure to understand the power provided by a modification is, in a sense, a weakness in the understanding of the stakeholders. Fully exposing these benefits provides a much clearer and comprehensive view of the claims that can be made for software security enhancement. We observe that while uncovering unknown benefits is inherently difficult, applying our argument approach focuses a systematic discussion which aids in revealing unanticipated benefits.

For example, SLX is not designed to provide buffer overflow protections within a complex data structures (i.e., buffer overflows within C structs). This deficiency was known and expected, but in trying to find if any argument can be made for intra-structure buffer overflow, some potential benefits were uncovered. C compilers frequently flatten structure data types, which essentially makes the structure semantically non-existent as long as the structure is not passed to other functions by address. SLX can therefore mitigate attacks within structures in these scenarios.

In other words, while SLX is not designed to mitigate intra-structure overflows, SLX can still provide some security protections in certain scenarios. How often these scenarios occur is an area for further study.

8.4 Results of Argument Limitations

The observed assurance case benefits described in the previous section were primarily the result of the systematic decomposition of the stakeholders' needs and the properties SLX is meant to provide. These observations suggest that the fitness argument provides necessary support for SSM at least in terms of the argument form, and in the ability to expose modification weaknesses and benefits.

The applicability of traditional assurance cases to SSM was limited, however, when considering the manner in which attack class mitigation is argued. The primary negative result was:

The variability in how SLX will modify software was difficult, if not impossible, to express with traditional argument structures.

This result is a consequence of the general approach used in the case study for argument development: recall that a single argument was developed to describe the acceptability of any SLX

modification. Traditionally, to construct an argument for a single instance of modified software, we would have to know exactly how the modification was applied. For example, for SLX modifications we would need to know exactly how many functions had been modified, and with what kind of protection mechanisms and configurations.

The issue with an SSM process is that during argument development, how SLX modifications will be applied is not known and can vary. To resolve the limitations of traditional assurance argumentation techniques, the concept of selection argumentation (Chapter 6) was invented. Application of selection argumentation is described in the next section.

8.5 Applying Selection Argumentation

This section describes example applications of selection argumentation within the SLX argument. To illustrate the use of selection argumentation, we limit the illustration to the two attack classes resulting from division of the "current frame data" attack class shown at the bottom of Figure 8.3[1]. We also limit our illustration specifically to the SLX canaries for these attack classes. The primary argument used for illustration is shown in Figure 8.4.

8.5.1 Function Variability

SLX modifications not only vary in how they are applied for each instance of SOUP, they also vary for individual functions within the same program. To development of the SLX argument, the argument must express the variability of modifications per function and the mechanics for selecting functions to modify.

In Figure 8.4, Goal 13.1 expresses a general function claim using entity abstraction (part of selection argumentation). The sub-argument for Goal 13.1 contains further variability described in subsequent sections. Obligation O1 expresses the general decision process for selecting functions

[1] Although the subdivision of attack classes increases the complexity and size of the argument, each attack class is mitigated using the same SLX mitigation techniques. While each mitigation argument can be subtly different, the same general approach for constructing the argument is used.

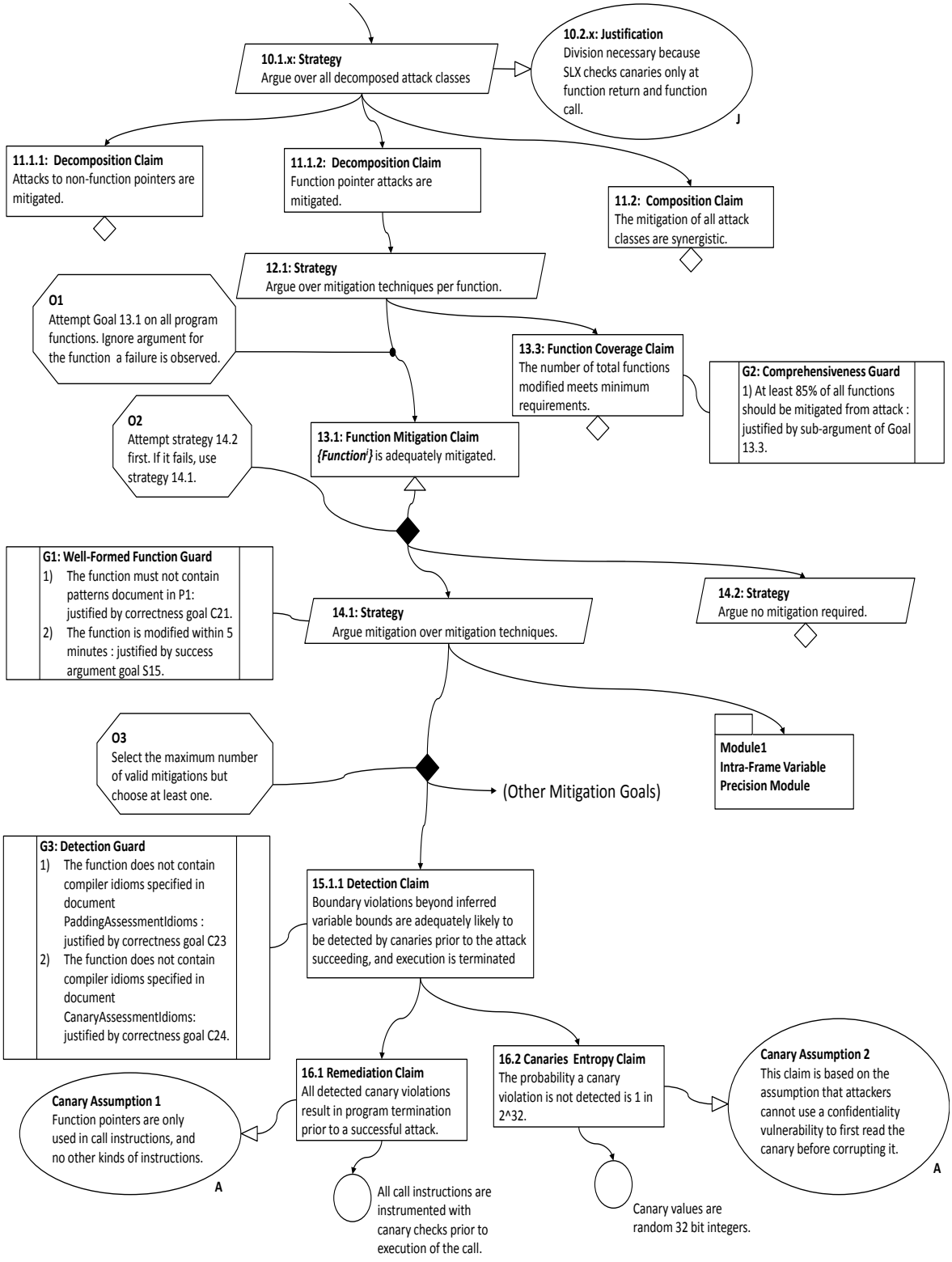


Figure 8.4: Example use of selection argumentation

to modify at this level in the argument. In this case, the decision process is to attempt to modify every function. If modification is not possible, the function is ignored. Criteria for when a function should not be modified are expressed through argument guards in the sub-argument for Goal 13.1 and the selection argumentation mechanics.

8.5.2 Minimum Acceptability Requirements

The original implementation of SLX performs a "best effort modification", i.e., SLX modifies as many functions as possible, and no consideration is given as to whether the result is tolerable. While stakeholders could accept a best effort approach, the argument becomes less compelling. For example, in the worst case, stakeholders must consider no modification whatsoever as an acceptable choice. The lack of a compelling argument forces us to consider the minimum acceptable modification requirements (i.e., the weakest acceptable modification). These requirements must be assessed when a modification is finalized.

In Figure 8.4, Goal 13.3 is added arguing that minimum modification requirements are met. SLX was not originally developed with minimum modification requirements in mind, so this goal is left undeveloped; however, for illustration, Guard G2 is applied to Goal 13.3. The premise of G2 is at some point the minimum modification requirements must be vetted, and conceptually these requirements guard Goal 13.3. For illustration, we invented a threshold number of functions that must be modified (85%). G2 indicates that determining if the modified program meets the 85% function modification threshold is not known a priori, and will require assessment for Goal 13.3 to be valid. Vetting minimum modification requirements specified in the guard would best be performed during the SSM finalization phase when a complete modification can be reviewed.

8.5.3 Mitigation Without Modification

A function can be adequately mitigated if the attack class does not pose a credible risk for the function or SLX modifications have adequately neutralized the threat. In practice, modifying the

software is risky; hence, arguing mitigation without performing any modification is preferable. The argument must therefore express a selection between these two alternatives.

In Figure 8.4, Strategy 14.1 corresponds to arguing mitigation by modifying the software and Strategy 14.2 argues mitigation because there is no credible threat. Obligation O2 expresses the preference between these two strategies (i.e., to prefer attempting Strategy 14.2). In our implementation of SLX, we did not explicitly consider the scenarios justifying no mitigation is necessary; hence Strategy 14.2 is undeveloped.

One potential check could involve determining if the number of variables detected for the function is 0 or 1. For an attack within the given attack class to be successful, there must be a vulnerable buffer and a target to exploit within the stack frame (specifically a function pointer). The minimum number of variables on the stack for the attack to be relevant is therefore 2. Only if Strategy 14.2 is not possible is an argument developed based on SLX modifications[2].

8.5.4 General Mitigation Restrictions

To decrease the risk of generating a modification that breaks expected program behavior, SLX performs several assessments to verify if a function is "*well-formed*". A well-formed function is a function that does not appear to have instructions that are known to result in incorrect SLX modifications and can be modified within a set amount of time. The rationale for these restrictions would be found elsewhere in the SSM assurance case; however, to make these crosscutting restrictions clear within the argument, the general well-formed function restrictions are expressed as in a guard (Guard G1) restricting the selection of Strategy 14.1.

8.5.5 Mitigation Variability

Strategy 14.1 argues mitigation by arguing over all for each function all three of the SLX mitigation techniques (canaries, variable padding, and variable reordering). The assessment mechanisms in

[2]Because the argument includes mitigation arguments for many attack classes, the lack of an SLX argument for one attack class does not necessarily suggest SLX modifications have not been applied to mitigate other attack classes.

SSM can invalidate any of the SLX mitigation techniques meaning any combination of mitigation techniques is possible. The argument must therefore express this variability and the process for selecting mitigation techniques. Additionally, stakeholders must consider the ramifications of limited or partial application of mitigation techniques. For example, stakeholders might tolerate a few functions not having canaries applied, but if all functions do not have canaries, they might consider this unacceptable.

In Figure 8.4, Obligation O3 documents how invalidating any of a mitigation techniques should be handled. In the latest implementation of SLX, the approach is to simply combine as many mitigation techniques that can be validated for each function, i.e., the maximum number of mitigation techniques that can be applied is considered valid. If no mitigation techniques are validated, O3 will not be able to choose a valid alternative, resulting in failure propagation described by the selection argumentation mechanics (see Chapter 6).

To restrict the overall variability of modifications, stakeholders can use Goal 13.3 (as described above) to specify minimum modification requirements for each function.

8.5.6 Canary Restrictions

The application of canaries allows for the detection of attacks at run-time; however, canaries cannot be reasonably applied in all cases. SLX performs assessments to verify if canaries can be applied without resulting in unintended program behavior. The assessments restricting the use of guards are specified in Guard G3. G3 specifies two assessments that must be performed. Both of these assessment reference compiler idioms that the function must not have in order to safely apply canaries. Rationales for these restrictions would be found elsewhere in the assurance case in a correctness argument.

8.5.7 Alternative Detection Argument

The mitigation argument for canaries in Figure 8.4 (Goal 15.1.1) is relatively simple because of the nature of the given attack class decomposition. The simplicity of constructing the argument

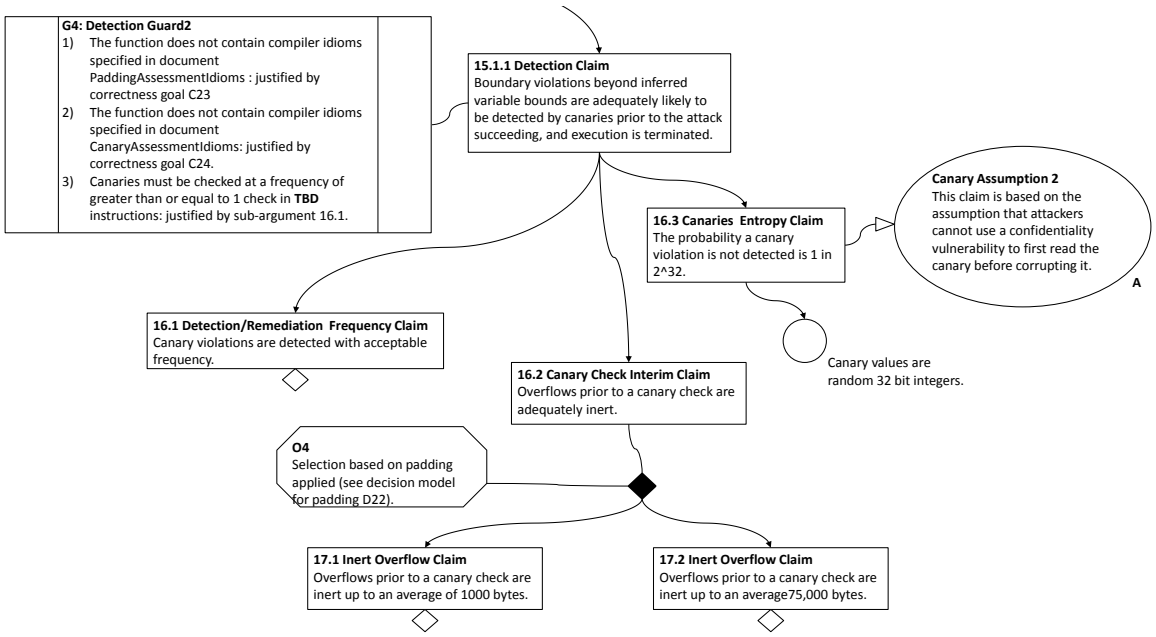


Figure 8.5: Canary argument for attacks targeting data other than function pointers

for this attack class lead to differentiating attacks to function pointers from attack to all other data within the stack frame (i.e., attacks to non-function pointers argued under Goal 11.1.1). Figure 8.5 illustrates the mitigation argument for canaries that would appear under Goal 11.1.1.

Because of how SLX was implemented, no argument can be made that guarantees a canary check will occur before data that are not function pointers are accessed. Instead, the detection of corrupted canaries occurs based on the frequency at which the canaries are checked. Goal 16.1 in Figure 8.5 expresses a claim about the frequency of canary checks, but the argument is undeveloped because we have yet to determine what will constitute an acceptable frequency.

Since the frequency cannot be known until an instance of software is provided, the frequency will need to be determine/assessed during modification. Guard G4 specifies (in addition to the general canary constraints) a constraint to validate the frequency is acceptable when the function is modified. Since we have not established what the acceptable frequency is, the frequency constraint is "To Be Determined" (TBD).

Even if the frequency of canary checks is tolerable, a claim can be made that the combined use of padding with the canary will render some linear buffer overflows inert until the canary

violation is detected, i.e., if the overflow does not exceed the boundary of the padding added by SLX. Goal 16.2 is added to address this scenario. The SLX mechanism for adding padding will choose a different padding amount depending on characteristics of the software, indicated by Goals 17.1 and 17.2. Obligation O4 references the decision process (found elsewhere) for choosing the amount of padding. The exact decision model for padding could be repeated in O4; however, in this instance we chose to use semantics of product line argumentation. Specifically, the decision affecting padding can be found in the mitigation argument for padding. Obligation O4 makes references to that more elaborate decision process, and states that the result of that decision will affect the decision between Goal 17.1 and 17.2.

8.5.8 Variable Boundary Precision

All SLX mitigation techniques rely on the precision of detected variable boundaries. SLX chooses between four naïve methods to derive variable layout based on accesses into stack memory: AOI, SOI, DOI, and ESI (see Chapter 7). The choice between these alternatives have an affect on assurance claims but the choice between these alternatives varies from function to function. In Figure 8.4, the variable boundary precision argument is extracted into a reusable argument module[3]. The detailed argument for Module1 is shown in Figure 8.6.

Selection argumentation allows the argument to express the selection process between the variable inference alternatives. Obligation O5 expresses the preference between alternatives. In this case, the preference is to begin with AOI then to proceed to the next inference containing the most variables. For the given attack class, at least 2 variables must be detected to make any security claims[4]; hence, ESI is not a valid alternative in this module. Each variable inference is guarded by identical testing procedures, specified in Guard G5.

[3]GSN module extensions are further described in the GSN community standard [19].

[4]The decomposition of attack classes makes the distinction between data targeted by an attacker that is within the currently executing stack frame (intra-frame attacks), and data that is outside an executing stake frame (inter-frame attacks). The decomposed attack class used for this illustration focuses on attacks to data within an executing stack frame. If one or fewer variables are found, the given attack class decomposition cannot be mitigated.

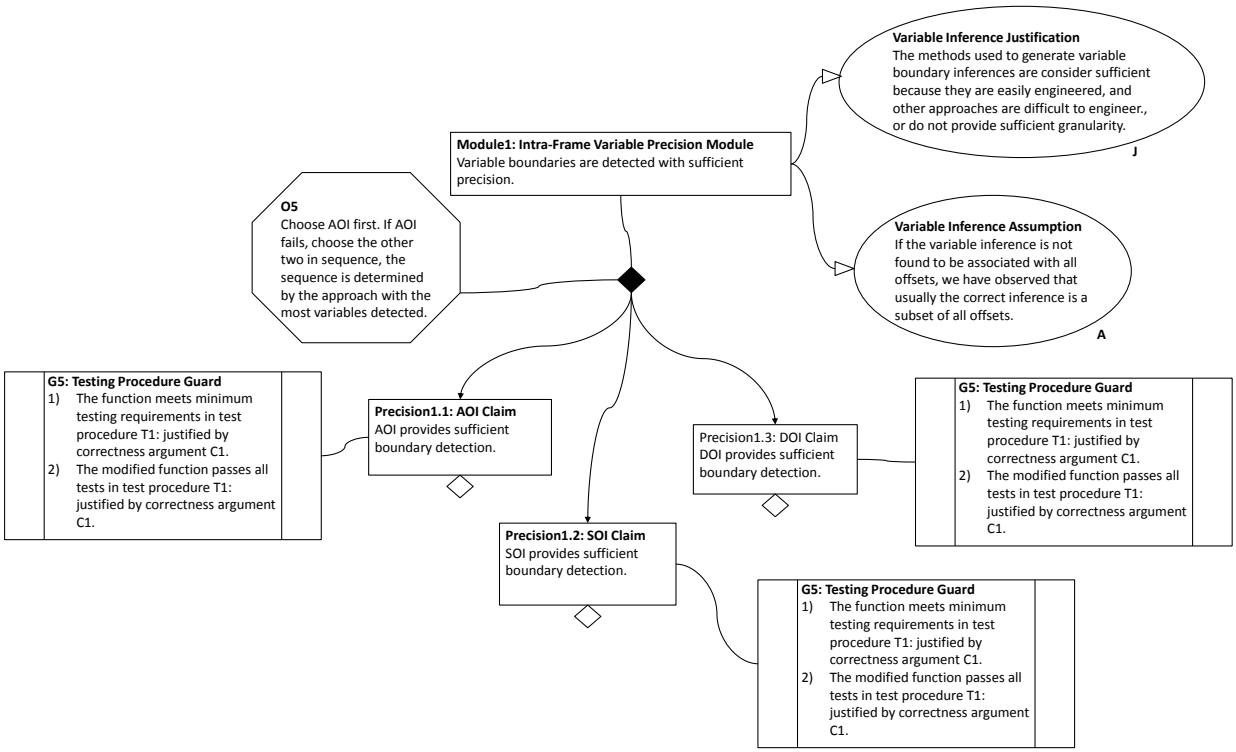


Figure 8.6: Module1: intra-frame variable boundary detection precision

The argument does not proceed further as we lack a convincing argument for any one of the variable recovery mechanisms. The lack of a compelling argument does not suggest we did not have intuition as to why any one of these techniques might work. Rather, trying to make a definitive and compelling argument based upon our intuition and ad hoc evidence was not possible without a significant research activity on variable boundary recovery.

8.6 Selection Argumentation Results

This section describes the results of applying selection argumentation. These results are summarized in Table 8.2 and further described in the following subsections.

Result	Summary	Section Containing Further Details
A success argument is necessary.	Discovery that a fitness argument is not sufficient for an SSM assurance case. A success argument is also required.	Section 8.6.1
Selection argumentation specifies SSM.	Discovery that selection argumentation notation and mechanics can be used as a specification for an SSM process.	Section 8.6.2
SSM assurance cases raise serious engineering challenges.	Discovery that SSM assurance case development raises questions about acceptability and completeness of the SSM implementation, e.g., the completeness and validity decisions and alternatives.	Section 8.6.3
Assurance case instances might require further assessment.	Discovery that assurance case instances derived from the mechanics of selection argumentation might require further assessment outside the SSM model.	Section 8.6.4
The criteria for which variable arguments are applicable.	Discovery of when the use of variable argument structures is appropriate within the argument.	Section 8.6.5
Readability of the argument is negatively affected.	Discovery that the visual complexity of the argument increases with the use of selection argumentation, potentially affecting readability.	Section 8.6.6

Table 8.2: Summary of selection argumentation results

8.6.1 The Success Argument

The constraints limiting the resources consumed by the SSM process have a direct impact on how modifications are produced. These constraints must therefore be made explicit within argument guards; however, the fitness argument does not provide any justification or rationale for these constraints. Initially, the justifications for resource consumption constraints were either not given explicitly or referenced within ad hoc documentation.

This approach did not provide a clear indication about how the SSM process restricts resource consumption when generating modifications. To provide better clarity, we adopted the use of success arguments from Assurance Based Development [31] within our concept of the SSM assurance case. Success arguments explicitly and rigorously document the resource consumption limitations of the

SSM process. By including success arguments in the SSM assurance case, all guards constraints are justified by components of the SSM assurance case, either in the success argument or the fitness argument.

8.6.2 Implementation Specification

As the SLX argument was constructed, the SLX process was conceptually abstracted into its underlying specification. The specification indicates:

- where decisions need to be made,
- the alternatives associated with each decision,
- how decisions are made and validated, and
- why each alternative is acceptable.

If developers were to begin by building the SSM assurance case, not only would they have identified weaknesses requiring attention before implementation, the developers would also have a road map to guide the development of the SSM process.

The mechanics of selection argumentation provide a general specification for the implementation of an SSM process. The exact manner by which the argument mechanics are realized in an instantiated SSM process are not restricted. Fundamentally, the exact implementation is not important so long as the mechanics within the argument are fully realized. The SSM process model is beneficial to developers to interpret and filter the mechanics in the SSM assurance case specified by selection argumentation.

8.6.3 Engineering Concerns

A benefit of selection argumentation is that once developers have specified decision models, alternatives, and guards, reviewers can easily understand the mechanics for generating a modification, and

recover an assurance case instance by obeying the argument mechanics. The SSM assurance case does not, however, answer the following questions:

- Are all decision models appropriate?
- Are there sufficient alternatives under each decision point?
- Are all possible assurance case instances acceptable?
- Will the mechanics specified by the argument produce acceptable modification (a complete assurance case instance) with a sufficiently high probability of success (i.e., are the selection mechanics appropriate for the given operating context)?

These questions are assurance concerns relating to high-level development concerns about the engineering and instantiation of the SSM model. An instantiated SSM model is the result of an engineering process of some kind, and can therefore be conceptualized as a software product. We can therefore apply any traditional engineering paradigm to assess and mitigate high-level developmental concerns, as described above.

High-level development concerns are important to address but exceed the scope of this dissertation. The main focus and scope of this dissertation is based the "product" an SSM instantiation is meant to generate (i.e., SOUP modifications), and the costs and constraints associated with generating that product. Regardless of what development paradigms are used in instantiation of SSM, the development activity should center around the construction of the SSM success and fitness arguments and using those arguments to realize the SSM process. In this dissertation, we assume that high-level development concerns have already been addressed in some manner, and instead focus on the form and mechanics of components within SSM. We leave further investigation into larger SSM engineering principles for future work (see Chapter 11).

8.6.4 Assurance Case Instance Assessment

Individual decisions within the argument can be easily understood and scrutinized. As more decisions are added, however, and the complexity of interactions between decisions increases,

we cannot easily understand the argument as a whole. For example, because of the fallacy of composition, we cannot always rely on our understanding of the components of the argument to translate into an understanding about the composition of all argument components. Without a complete understanding of all arguments that can be generated (all assurance case instances), we must consider the possibility that an assurance case instance is generated that is unacceptable.

Compositional correctness of arguments is an area of future research and might be addressed in the instantiation of the SSM model; however, a key result of this case study is:

Doubts about an assurance case instance are the same as doubts about any assurance case for any software system.

Residual doubts about arguments always remain, and might require assessment on an individual argument bases. Additional review and assessment of individual assurance case instances might therefore be unavoidable.

Argument assessments can be performed manually by experts or assessment can be performed through automated mechanisms. In Chapter 5, an argument assessment framework is proposed that could be adopted for the automatic assessment of assurance case instances. Using the framework, confidence would first be assessed by experts for individual assertions within the SSM fitness argument. When an assurance case instance is generated and argument components are combined and validated, the confidence metrics associated with chosen argument structures would be combined and computed using stakeholder-defined confidence arithmetic. Failure to meet defined confidence thresholds would invalidate the assurance case instance.

Extending our assessment framework for automated argument assessment is an area for future work. Automated or manual assessment of assurance case instance could be considered as part of the SSM process finalization phase. For simplicity of this dissertation, however, we consider the assessment of assurance case instances a separate activity performed within a larger SSM deployment cycle and is left for future work (see Chapter 11).

8.6.5 Applicability of Variable Arguments

Initially in this study, a fitness argument was constructed to intentionally hide the variability of SLX modifications, i.e., the argument did not prescribe any variability or complex decision processes. The intuition was to make a single yet general argument that would applicable to all SLX modifications in its original and unaltered form. The problem with an argument of this form is that the argument would have to express the *weakest acceptable argument* (i.e., the minimum modification requirements stakeholders will accept).

While developing a weakest acceptable argument is a valid approach for arguing about software modifications, SSM is applied to produce modifications within a space of acceptable solutions. If a weakest acceptable argument is used, there is no indication where any given modification falls within the acceptable solution space. Further, there is no indication or rationale as to why SSM generates and assesses modifications at all. Selection argumentation provides a much more detailed view of SSM activities, to an extent that allows the assurance case to serve as a specification for implementation.

In principle, selection argumentation should be used if stakeholders care about differentiating between weaker modifications and stronger modifications. Argument developers must use their own judgment to determine when an argument can be made in a general sense, or if selection argumentation is best applied.

8.6.6 Readability

Selection argumentation clarifies the mechanics of constructing an assurance case instance, but also negatively influences the readability of the argument. Additional guard and obligation elements clutter the argument structure and can be visually unappealing. While readability is a detractor to the use of selection argumentation, readability can be improved with proper argument development tools. With tool support, guards and obligations can be hidden when not required, and organized into tabular structures and databases for ease of reference.

8.7 Results Summary

The goal of this case study was to answer the questions presented in Section 8.1. The answers to these questions are summarized in the following subsections.

Question 1: Feasibility Results

For an SSM assurance case to be feasible, it must provide sufficient expressive capability for SSM and be possible for engineers to develop. Using traditional assurance case methods alone does not provide sufficient expressive capabilities to support SSM; however, the use of a success argument and the application of selection argumentation allows the SSM assurance case to overcome this limitation. Specifically, the use of the success argument with selection argumentation allows the assurance case to express both the solution space and the criteria for navigating the solution space within the argument structure itself.

While selection argumentation and the use of a success argument does increase the complexity of the argument, we did not observe that the added complexity unreasonably increased the difficulty of constructing the argument. In fact, the added complexity was trivial compared to the larger development challenges of:

- reverse engineering an argument from a completed SSM process,
- producing a compelling argument structure, and
- generating evidence when sufficient evidence is lacking or nonexistent.

These more significant challenges are known challenges in applying assurance cases generally. The first challenge can be addressed by developing the argument prior to or in parallel with the development of the SSM model [23]. The latter challenges are fundamentally necessary to overcome in order to provide assurance for any software system. Although developing a complete assurance case can require a significant engineering effort, to date, many successful production-level assurance cases have been produced [17]. We therefore conclude that the construction of the SSM assurance case is feasible.

Question 2: Practicality Results

In order for the SSM assurance case to be practical, it must allow reviewers to understand the acceptability of SSM modifications. In this study, we have demonstrated how the development of assurance case can expose the benefits and weaknesses of modification techniques. Further, by including within the SSM assurance case a success argument and selection argumentation, we have demonstrated how the SSM assurance case captures the desired mechanics for developing acceptable software modifications.

The SSM assurance case serves as an assurance blueprint for constructing acceptable SOUP modifications. Reviewers can traverse the selection argument mechanics and the corresponding arguments to understand the decision, validation and refinement processes and why these activities are performed. In this respect, the SSM assurance case does facilitate understanding of SSM modifications.

Individual decisions and validation procedures can be easily scrutinized but it is fundamentally challenging to understand if the sum of all decisions will result in an acceptable modification, i.e., will any produced assurance case instance be acceptable? While additional assessments might be required, either on the entire SSM assurance case or a generated assurance case instance, the practicality of the SSM assurance case is not invalidated. The SSM assurance case provides the general assurance structure for software modification. Without the SSM assurance case, no further assurance assessment would be possible. While additional research is necessary to address compositional correctness of arguments and other developmental concerns mentioned Section 8.6, the SSM assurance case provides the structure for further assurance assessment. We therefore conclude the SSM assurance case is practical.

Chapter 9

Case Study: Exploring the Applicability of SSM

The previous two case studies have evaluated and scrutinized the feasibility and practicality of SSM with respect to:

1. the process mechanics and
2. the SSM assurance case.

While the results of these studies were positive, both evaluations were limited to a single specimen instantiation of the SSM model (i.e., SLX). Based on only one target of evaluation, there is no clear indication that the SSM model will be applicable in other scenarios or how engineers would determine if the SSM model were an appropriate solution.

Ideally, further evaluation of SSM would investigate the broader practicality and feasibility of the SSM concept. A rigorous evaluation would require a statistically valid set of replicated trials across a range of operating contexts, stakeholders, and methods of modifying the software. Multiple characteristics about the general applicability of SSM could then be examined, such as:

- The general conditions where SSM is applicable and inapplicable.

- The benefits of applying SSM in terms of desired properties the software is meant to maintain (especially security).
- The effort in instantiating SSM.
- The performance of SSM modifications in real-world contexts over time.
- The effort in maintaining and updating SSM over time, and the conditions leading to the need to update SSM.
- The guidelines for applying SSM.

The scope of the above evaluation is not feasible within this dissertation. Instead, this chapter provides a more limited and simplified evaluation of SSM. This chapter evaluates the practicality of SSM by addressing the research question:

Can SSM be applied to other modification approaches and how?

To limit the scope of this study, the applicability of SSM is examined for a new security modification technology developed by others independently of this research. Further, the development an assurance case is assessed as a means for both examining and demonstrating the applicability of SSM.

9.1 Case Study Overview

The goal of this study, as mentioned above, is to evaluate the practicality of SSM with respect to the applicability of the SSM model. In this study, applicability is not examined by creating and evaluating multiple instantiations of SSM; rather, this study addresses the following research questions for a single target of evaluation (i.e., a single modification technique):

1. Can the development of an assurance case be used to determine the applicability of the SSM model?

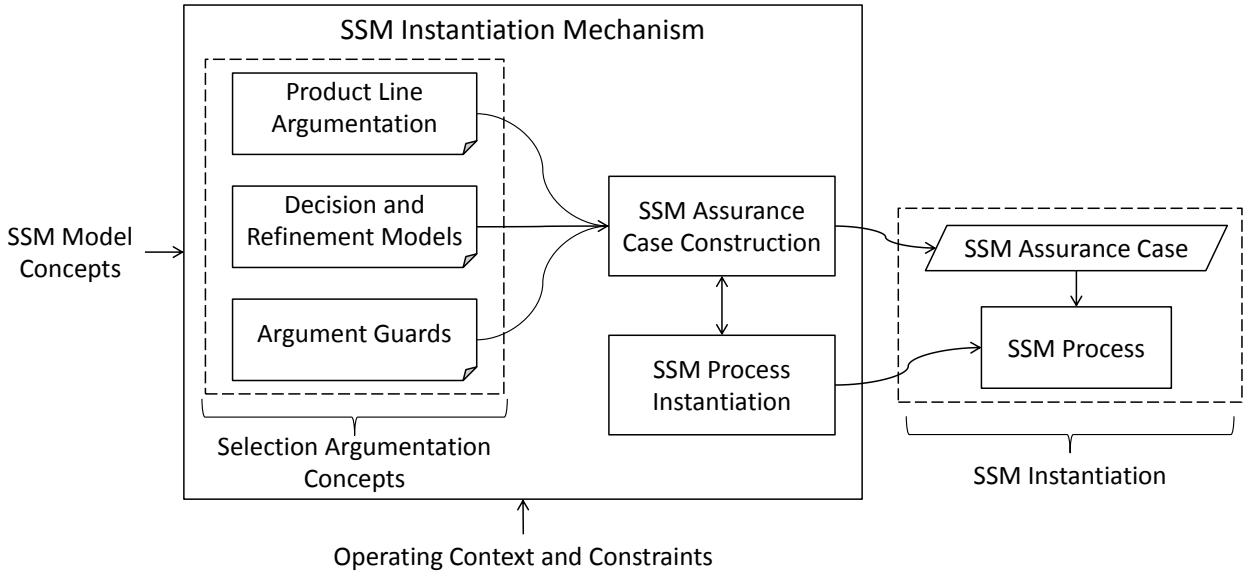


Figure 9.1: SSM instantiation concept

2. How is selection argumentation applied within the assurance case to specify SSM process mechanics?

The rationale for focusing on the development of an assurance case is based on the results of the previous case studies. The previous studies suggested that:

- the development of an assurance case provides benefits in reasoning about software modifications, and
- selection argumentation can be used to specify desired SSM process mechanics within the assurance case.

Based on these observations, this study uses the development of an assurance case as a method for examining and demonstrating the applicability of SSM for a single modification technique.

9.1.1 Preliminary Strategy and Results

The general strategy used in this case study for determining the applicability of SSM was to isolate areas within the assurance case that cannot be developed unless selection argumentation

Result	Summary
Acceptability of a modification technique is based on its application.	An assurance case cannot be developed for a modification technique itself but rather the assurance case is developed for the *software system* that has been modified.
Variability and uncertainty about software systems might inhibit assurance case development.	Evidence to support an assurance claim might be based on data about a specific software system. Since that data might be unavailable and uncertain, development of an assurance case might be restricted.

Table 9.1: Preliminary results summary

is applied. Since selection argumentation specifies an SSM process, demonstrating the need for and application of selection argumentation also demonstrates the need for and application of SSM. Further development of an SSM could then be accomplished by interpreting the resulting SSM assurance case, as illustrated in Figure 9.1.

Determining the precise conditions where selection argumentation is applicable is part of the experiment of this case study. In preparation for this study, we discovered two additional results that were used to focus the experiment. These results are summarized in Table 9.1.

In practice, a modification technique will be applied across a range of software (as illustrated in Figure 9.2). To develop a single assurance case to describe the acceptability of all modified software systems is inherently problematic since the assurance case must be sufficiently general to be applicable for all modified software systems. The goal of this case study is to develop a sufficiently general assurance case, but we anticipated that some evidence necessary to complete the assurance case will be based on the characteristics of a specific software system. Such evidence is missing, uncertain, and likely to vary for each software system. For this study, examination of missing, uncertain and variable evidence was used as the driver for determining the applicability of SSM.

9.1.2 Case Study Characteristics

This case study is based on the following premises:

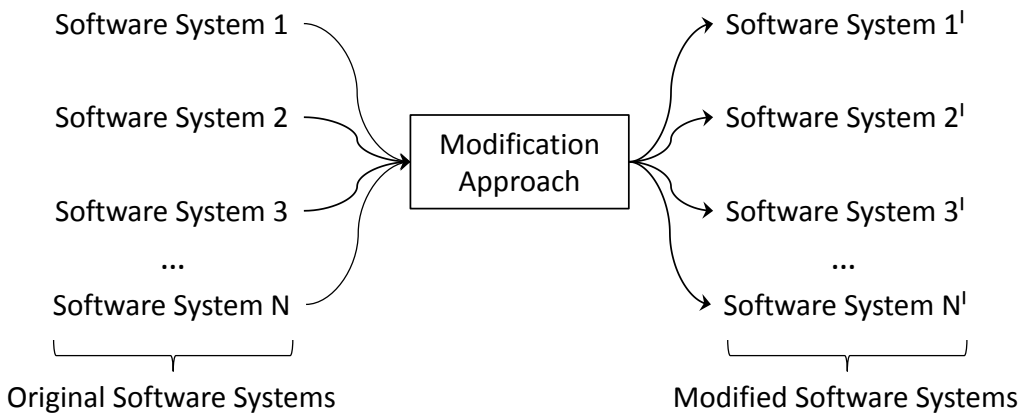


Figure 9.2: Application of software modifications in practice

- We take the position of stakeholders of an organization using a large amount of software.
- The stakeholders consider their software at risk of attack and look to apply defensive technologies in the form of security-enhancing software modifications.
- The stakeholders have already identified their specific security threats and have already chosen a target defense technology.
- All software currently in use and any software that will be used in the future will be modified using the target defense technology.
- Development of an assurance case is used primarily to understand how the target defense technique will be practically applied.
- Development of the assurance case proceeds without knowing if SSM is applicable.
- The experiment of this study is to determine the applicability of SSM by assessing and developing the assurance case.

For simplicity, the stakeholders have already identified a need for applying software modifications and have also identified the specific security threat. To avoid bias, a target modification technology is chosen that was invented, developed, and put into practice independently of this

research. The targeted modification technique is designed to defend against operating system (OS) command-injection attacks and is referred to as *Software DNA Shotgun Sequencing* (S^3) [84].

This study begins primarily as an evaluation of S^3 through the development of an assurance case. Software modifications (like S^3) are intended to improve the quality of the software. The development of an assurance case provides a rigorous justification as to why the modifications are acceptable. Assurance case development therefore provides a common and reasonable (i.e., practical) starting point for engineers regardless of their particular operating context and needs or whether the SSM model is applicable.

9.1.3 Scope

This study focuses on the initial pragmatics of applying SSM. The target of evaluation (S^3) was out of our direct control for this study. Evidence necessary to complete the assurance case was often limited or unavailable. Additionally, S^3 was implemented prior to our study and not developed with respect to the SSM model or an assurance case. To complete the S^3 assurance case would require co-development between argument engineers and the S^3 development team.

This study demonstrates the first iteration of an SSM co-development activity (i.e., the first iteration of an Assurance Based Development (ABD) activity[1] [30, 31, 23]). The assurance case is developed to provide a *compelling* argument justifying the use of S^3 for any given software system. Since S^3 was not developed to meet the needs specified in an assurance case, a compelling argument might specify claims about S^3 that are not yet supported. Further development iterations of S^3 would then be required to complete the argument. These development activities are not examined in this case study.

Since SSM can be arbitrarily complex (i.e., stakeholders can specify any number of alternatives and validation activities), this case study is limited to demonstrating the initial activities in applying SSM. This case study highlights within the assurance case where SSM might be applicable and could be further developed. The result of this approach is a partial SSM assurance case with initial

[1]The concept of Assurance Based Development (ABD) is further described is given in Chapter 10.

SSM mechanics specified (i.e., initial application of selection argumentation). This study assumes the connection of the SSM assurance case mechanics to an SSM process is apparent (see Chapters 6 and 8). Further development of the SSM assurance case and instantiation of an SSM process are left for future work.

9.2 Target of Evaluation: S^3

Software DNA Shotgun Sequencing (S^3) [84] is a novel approach to thwarting command-injection security attacks. S^3 is designed to operate with no information about the software other than the binary form. The concept is to make transformations to the binary program to prevent exploitation of vulnerabilities that might be present but are unknown.

Vulnerabilities that can result in operating system (OS) command injection attacks (a special case of command injection attacks) are the second entry in MITRE's 2011 CWE/SANS list of top 25 most dangerous software errors [65]. Operating system commands are the means whereby programs make requests for operating system services, including file and network operations. If an attacker can compromise an application and issue arbitrary commands to the underlying operating system, the damages could be extensive. A network-facing server running with high privileges could be attacked, for example, with the potential for the loss or leakage of extensive sensitive data.

The OS commands of interest are those that contain parameters derived from outside the software. For example, an application can issue an OS command to output the contents of a file identified by the user:

```
char[MAX_LEN] cmd = "/usr/bin/cat";
fgets(file, BUFFERSIZE, stdin);
...
strcat(cmd,file);
system(cmd);
```

If instead of providing a file name, the user provides "`; rm -rf /`", the cat command will execute with no file, and consequently fail, followed by the rm command. The rm command would proceed to delete the entire root partition of the file system if the software were running with administrator privileges[2].

Clearly, the solution cannot be to deny all OS commands. Instead, many modern defenses rely upon taint analysis, i.e., determination of whether information to be used in an OS command can be trusted. Prior to issuing a security-sensitive OS command, the command is first checked against its taint markings to ensure that critical parts of the command are not tainted.

Taint analysis can be based upon either (1) positive taint in which trusted data is analyzed or (2) negative taint in which untrusted data is analyzed. Analysis is performed either by (1) tracking the flow of data (and hence taint) from an external source as the data propagates through a program to a security-sensitive operation, or by (2) inference in which the taint is inferred in some way.

Applying existing taint tracking techniques to binary programs is problematic, because the execution-time overhead can be prohibitive [85]. The goal for S^3 is to provide effective, low overhead taint analysis that can be applied to binary programs. The approach used by S^3 is positive taint via inference. Prototype implementations of S^3 have been developed for binary programs running on Intel's X86 architecture for both OS command injection attacks and SQL injection attacks. In this case study, the focus application of S^3 is for OS command injections specifically.

9.3 The S^3 Approach

The S^3 approach is summarized here. Full details about the approach and experimental results are available from Nguyen-Tuong et al. [84].

The S^3 attack detection architecture is summarized in Figure 9.3. S^3 is structured as five major components. The DNA Fragment Extraction component extracts string literals, i.e., DNA

[2]We note that the dangerous and common SQL injection attacks in which malicious parameters from outside the software are included in a command to an SQL database are non-OS command injection attacks. The techniques described here have been adapted for SQL injection attacks.

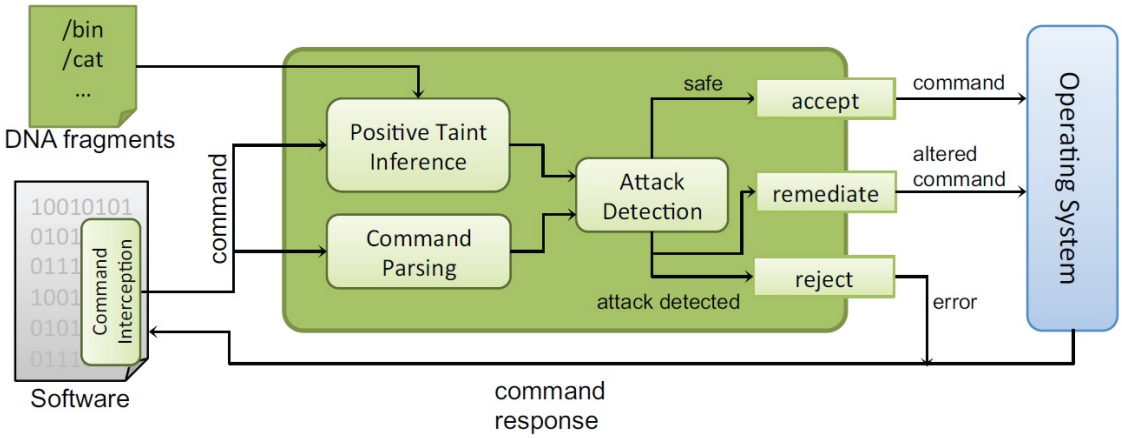


Figure 9.3: S3 architecture — figure taken from Nguyen-Tuong et al. [84])

fragments, from the binary program and the associated libraries. This analysis is done once, prior to program execution, and the analysis time is not part of the execution-time overhead. The Command Interception component intercepts security-critical commands generated by the subject program during execution so that the commands can be examined.

By matching the command against the extracted DNA string fragments, the Positive Taint Inference component determines which characters in the intercepted command should be trusted. Any unmatched character is deemed untrusted. Using DNA fragments native to the software to infer taint is a novel form of taint inference and one of the key contributions of the S^3 architecture.

The Command Parsing component parses an intercepted command to identify critical tokens and keywords (i.e., components of the command that can be used to achieve a malicious purpose). The Attack Detection component combines the output of the Positive Taint Inference and Command Parsing components to determine whether an attack has occurred. A command is deemed an attack if a critical token or keyword is not marked as trusted (i.e., the critical token or keyword is not found within the extracted set of fragments).

If S^3 detects an attack, the command is either rejected outright or altered before being passed to the operating system. The current prototype implementation simulates a failed command invocation by substituting an error code in place of the actual command.

S^3 meets the goal of operating with low overhead, but what about S^3's efficacy? The detection of malicious commands is based on inference, and two types of failure are possible:

1. a benign command could be inferred to be an attack, i.e., a *false positive*, or

2. a malicious command could be inferred to be benign, i.e., a *false negative*.

S^3 uses a positive taint inference based on the set of strings recovered from the subject binary program. The two types of failure noted above arise because the inferences and assumptions in S^3 are imperfect. The rate of false positives and false negatives depends upon the algorithms used in S^3 and on the specific program to which they apply.

The crucial importance of the assurance case for S^3 is to provide a framework for judgment about assumptions, inferences, the applicability of analyses, etc. For example, there is no way to guarantee the absence of false positive or false negative of failures. Thus the assurance case for S^3 documents the rationale for belief that S^3's efficacy is adequate to meet the needs of the stakeholders of a system to which S^3 has been applied.

9.4 Assurance Case Development

Recall that the development of the assurance case is used as an experiment in this study to determine if the applicability of SSM can be derived from the development of the assurance case. This section presents key concepts and aspects of the development of the S^3 assurance case.

9.4.1 Prerequisites

Our approach to beginning with the development of an assurance case is predicated on the conclusion that a fit-for-use and success argument must be constructed. This conclusion is based on the following prerequisites:

- Engineers must understand the concept of assurance arguments and the Goal Structuring Notation (GSN).

- Engineers must understand the concepts of fit-for-use and success arguments.
- Engineers must determine that multiple instances of SOUP will need to be modified.

In principle, engineers primarily need to concentrate on the development of the fit-for-use argument; however, once engineers determine they will need to modify multiple instances of SOUP, some justification is necessary to demonstrate resources used to modify each instance are acceptable. Consequently, a success argument is also necessary. Since in practice modifications are meant to be applied to a range of software (see Figure 9.2), we anticipate the need for a success argument is common.

The need to modify a large corpus of software also necessitates development of a "*prototype*" assurance case (discussed in Section 9.1). Rather than develop an assurance case for a specific system, the assurance case is developed to express the application of the modification in all cases. This approach is beneficial in that the resulting assurance case reveals the complete spectrum of issues associated with S^3 and provides a framework for developing instantiations of S^3 for specific systems.

The argument structure and general claims within the success and fit-for-use arguments are identical to those discussed in Chapters 3 and 4. Engineers then proceed to construct the success and fit-for-use arguments using traditional argument methods (i.e., without selection argumentation). For this study, we assume that engineers are at least aware of selection argumentation and the SSM model as presented in this dissertation.

9.4.2 Security Argument Structure

Development of the assurance case for S^3 primarily focuses on the development of fitness argument structure described in Chapter 4. Recall that the argument structure is deconstructed into four abstract levels:

- Level 1 - Fit for Use
- Level 2 - Attack Classes

- Level 3 - Decomposed Attack Classes
- Level 4 - Attack Class Mitigation

Here we summarize the top levels of the argument mostly in text form for brevity.

The fit-for-use level of the security argument argues over the security requirements for the subject system and other constraints. The general approach to argument development was to focus on the development of the security requirements argument. As the security requirements argument was developed, any constraints affecting security claims were questioned, such as:

- Fitness constraints (e.g., correctness and efficiency).
- Success constraints (e.g., the resources required to produce a modification).
- Confidence constraints (i.e., confidence in all argument assertions).

Both the success and fitness arguments were developed in response to provide assurance that the above concerns were addressed.

In the case of S^3, there is only one attack class of interest, OS command injections, and so level's 1 and 2 of the argument are simplified to that effect. The taxonomy used to determine the known attack details was the Common Weakness Elaboration (CWE) [42]. For OS command injections, the CWE of interest is CWE 78.

Level 3 of the argument structure (decomposed attack classes) separates the arguments for statically- and dynamically-linked binaries. For purposes of security analysis, these two cases are distinct, and the vulnerabilities are completely different because of implementation details specific to S3. In modern information systems, dynamic linking is much more common, and so the prototype S^3 argument is developed for this case only.

Recall that the basic taint inference in S^3 assumes that strings within the binary can be trusted and that OS command injection attacks will originate from externally supplied strings. Nevertheless, attacks could originate from internal strings, e.g., strings that were corrupted by a previous preparatory attack. Level 3 of the S^3 argument argues over these two cases although only the former

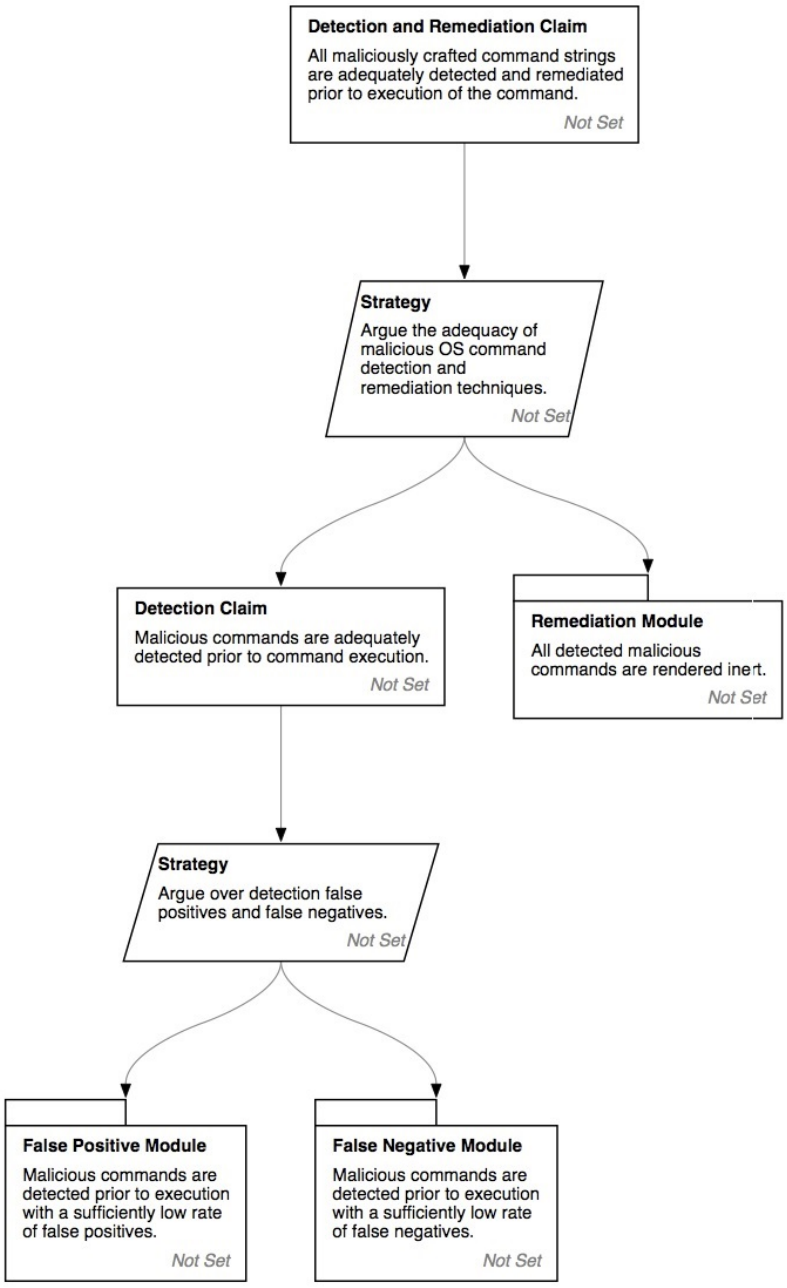


Figure 9.4: Detection and remediation argument structure

is developed. Finally, the level 4 sub-argument for this element of the S^3 argument begins with the claim (shown in Figure 9.4): *All maliciously crafted command strings are adequately detected and remediated prior to execution of the OS command.*

The argument structure used for this goal contains two sub-goals:

1. Malicious commands are adequately detected prior to command execution.

2. All detected malicious commands are rendered inert.

The argument structure used for the first of these two sub-goals is itself subdivided into two sub-goals:

1. Malicious commands are detected prior to execution with sufficiently low rates of false negative failures.

2. Malicious commands are detected prior to execution with sufficiently low rates of false positive failures.

The details of the sub-arguments for each these two goals are discussed in the following subsections.

9.4.3 False Negative Failures

Recall that, for S^3, a false negative is a failure to recognize a malicious OS command when a binary program protected by S^3 is executing — an extremely serious situation.

ALARP

Ideally, the false negative argument in the S^3 assurance case would compel belief that either:

- a false negative failure is not possible, or

- that the probability of a false negative failure occurring is below some prescribed threshold.

Clearly false negatives can occur with S^3, and so we cannot argue that a false negative failure cannot occur.

The S^3 technology prohibits comprehensive probability quantification, because several elements of the technology rely upon distributions that are completely unknown. We therefore can only argue that the risk associated with false negatives is acceptably low by arguing that risks are reduced *As Low As Reasonable Practicable* (ALARP) [15]. Determination of whether the level achieved is

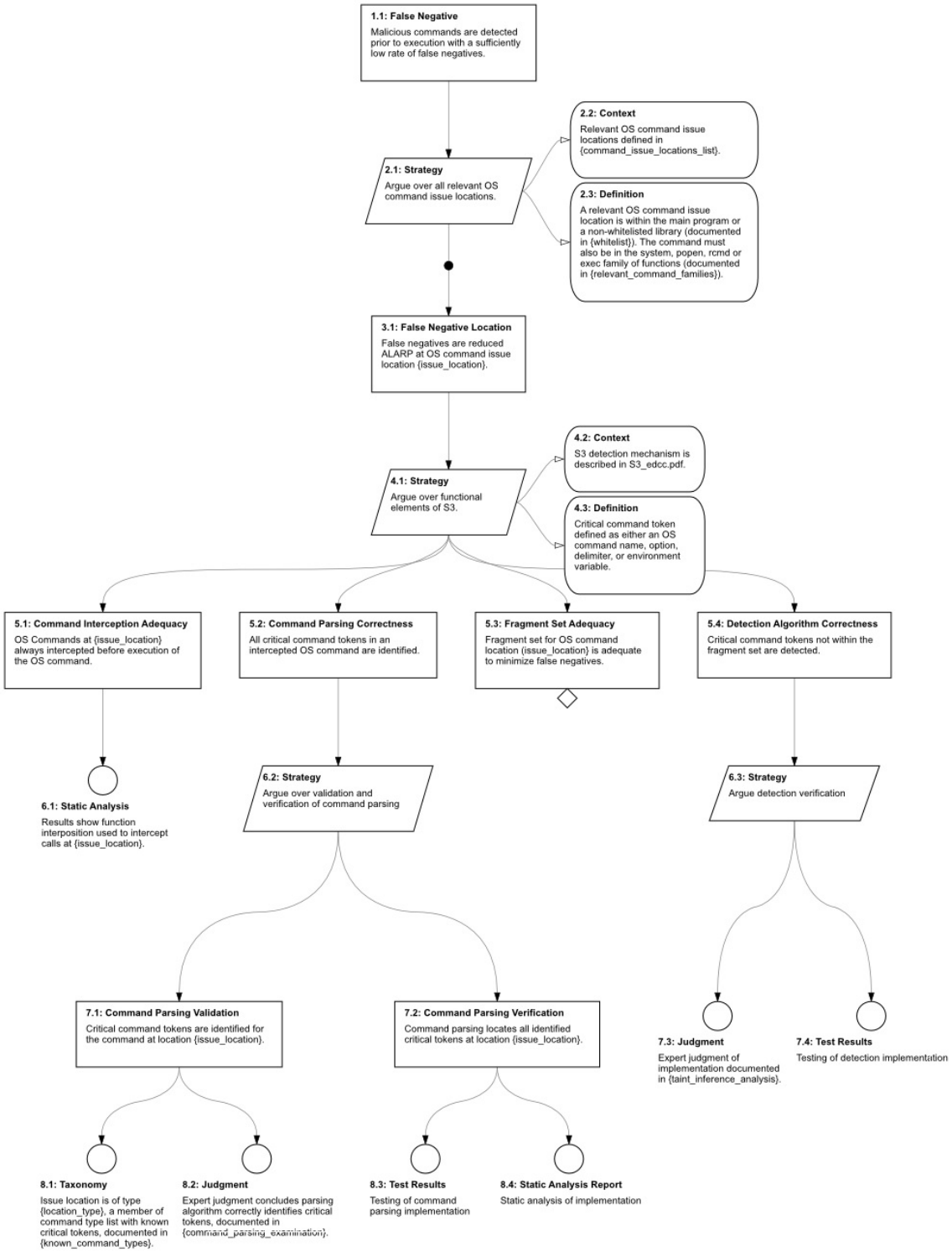


Figure 9.5: False negative argument summary

acceptable is then the responsibility of the stakeholders. The false negative ALARP argument in the S^3 assurance case is summarized[3] in Figure 9.5.

The false negative argument structure depends extensively on terminology specific to S^3. The original publication by Nguyen Tuong et al. [84] presents all of the S^3 terminology. Some of the key S^3 concepts are:

Critical Token: An OS command contains one or more tokens (in the sense of a language definition) that are considered critical, because they could be used maliciously. More specifically, S^3 defines command names, options, delimiters, and the setting of environment variables as critical tokens.

Fragment Set: The fragment set is a set of string literals, referred to as fragments, that are considered trusted if found within an intercepted OS command.

S^3 infers taint by matching tokens in an intercepted OS command to fragments. If the OS command contains critical tokens that are not matched to a fragment, the OS command is considered tainted. And tainted commands are interpreted as attacks.

Argument Granularity

The first strategy in the argument for false negative failures argues over all locations from which OS commands are issued. An argument about false negatives must demonstrate that, for all such locations within the application, the rate of false negatives have been reduced ALARP. This element of the argument is both important and subtle.

The essence of the issue is that the mechanics of an exploit and the associated cost of an OS command injection attack depend upon the memory location from which the OS command is issued, i.e., the circumstances of the command. For example:

- How a command is parsed might differ at different locations, i.e., what is considered a critical token might differ depending on the particular OS call.

[3] Arguments in this dissertation are summarized because the complete arguments are quite large.

- The contents of the fragment set should be customized to each location. For example, the files to which a call might need access might depend on the location from which the OS command is issued.

If the strategy were to argue about all locations that issue OS commands at once, the argument would assume the same command parser and fragment set in all instances. The lack of specificity presents more opportunities for false negatives. For example, fragments that might never be legitimately used at one location would have to be part of the fragment set if the fragment serves a legitimate purpose at any other location.

In principle, arguing about all locations from which OS commands are issued at once might be possible (or at least considered), but to do so would require that one determine that the associated argument is, in fact, identical for each location. Prior to completing the S^3 argument, we cannot be certain that a single argument is applicable to all locations, and that determination is program dependent. Thus, the S^3 false negative argument argues over all locations that issue OS commands, each with an individual false negative goal. If the argument were found to be identical for some subset (or indeed all) locations, the argument could be repaired later. In practice (as described below), we concluded that, for the most part, the argument at each location should differ.

The strategy that argues over each site that issues an OS command has to be in the form of a GSN pattern. We do not know prior to applying S^3 how many OS commands will be issued or their locations in any given application. Instead, we must use structural and entity abstractions typically used within GSN argument patterns [19] and within product line arguments [22] (see Chapters 2 and 6). In the S^3 argument, a black dot is used to indicate a repetition of the argument structure, and text within curly braces refers to entities that are dependent on the specific application of S3.

Arguing over OS command locations requires the definition of relevant locations. Stakeholders might consider applying S3 to all OS commands in an application, or they might trust some entities such as libraries like libc. In the current prototype implementation of S^3, the system trusts some libraries, and they are identified using a white list. In principle, there might be entities that stakeholders always prefer to trust across all applications, and this possibility is provided for

within the S^3 argument as a context item (see Figure 9.5). Similarly, all locations from which OS commands are issued within the target software must be identified, and this set is also referenced within a context.

For each of these contexts, strong confidence must be demonstrated that the context is appropriate, sufficient, and trustworthy. In the S^3 argument, separate confidence arguments are assumed (see Chapter 5). Confidence in the white list might be supported by evidence in the form of expert judgment. Confidence in locating all locations issuing OS commands might include some form of static analysis of the application.

False Negative Functional Hazards

At the next level of the argument, we argue that, for each injection location (of which there could be many), the associated rate of false negative failures is ALARP by arguing over four goals derived from the basic S^3 mechanism. These four goals are:

Command Interception Adequacy: OS commands at a given location are intercepted. If not, any associated attack(s) might not be detected.

Command Parsing Correctness: Given an intercepted OS command, command parsing identifies all critical tokens correctly. Since detection of attacks is based on analysis of critical tokens, failure to identify a crit-ical token could allow an attacker to inject a malicious command.

Fragment Set Adequacy The fragment set must have specific character-istics in order to properly imply trust. Fragments within the fragment set are compared to critical tokens in intercepted OS commands, and the parts of an OS command that match fragments are considered trusted.

Detection Algorithm Correctness As with the previous goal, the frag-ment set and the parsed critical tokens are used to infer taint, and so the associated algorithm must work correctly.

The sub-arguments for these four goals are summarized in Figure 9.5. Compelling evidence has not been obtained for the prototype implementation of S^3, and so we hypothesize feasible types

of evidence. The first and fourth goals could be solved by verification evidence. For example, the adequacy of command interception might be solved by evidence from static analysis of the transformation of the binary and the associated insertion of the necessary probes. Similarly, algorithm correctness might be shown by evidence from testing, static analysis, proof, or some combination.

The second goal, Command Parsing Correctness, is decomposed in the argument into two sub-goals:

1. Command Parsing Validation, and

2. Command Parsing Verification.

The first sub-goal refers to the need to identify the necessary set of tokens and the second to the need to identify the critical tokens in a specific intercepted OS command. In general, the evidence for the solution of each of these goals could derive from expert judgment, testing of similar programs, taxonomies developed separately, and so on.

The third goal is the most difficult of the three for which to develop a compelling sub-argument because of the speculative basis of the fragment set. The goal in its entirety is "Fragment set for OS command location $\{issue\_location\}$ is adequate to minimize false negatives." where adequate is a practical manifestation of accurate. The sub-argument associated with this goal is discussed in the next subsection.

Fragment Set Adequacy

Many different approaches to the sub-argument for the fragment set adequacy goal could be developed. The approach we use in our S^3 assurance case is summarized in Figure 9.6. We argue the following two sub-goals:

The Constituent Fragment Goal: The fragment set should only containing constituent fragments. A constituent fragment is a string containing a sequence of one or more critical command tokens for the given location in an acceptable order.

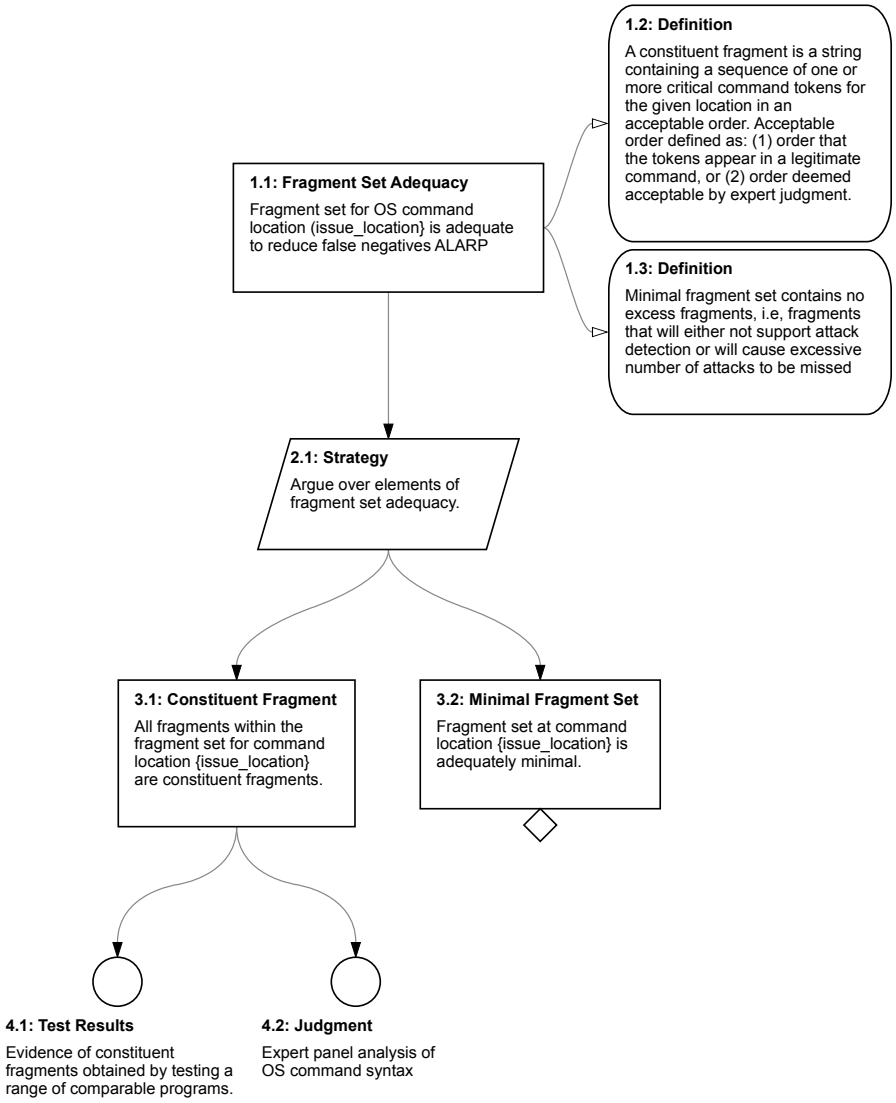


Figure 9.6: Fragment set adequacy sub-argument summary

The Minimal Fragment Set Goal: Even if the fragment set is shown to contain only constituent fragments, some fragments might not be effective (single characters for example). Such fragments can be thought of as "junk" DNA. In the limit, if there were too many such fragments, few or no attacks would be detected. We therefore stipulate that the fragment set must be adequately "minimal". How best to identify ineffective fragments is not presently known.

Evidence for these goals could take many forms. First, we note that acceptable order in the definition of constituent fragment might be defined to be one or more of the following:

- Any order.
- The order that the tokens appear in a legitimate command.
- An order deemed acceptable by expert judgment.
- An order determined by analysis of strings used in practice by similar programs.
- An order that could occur as a result of the control flow of the program that leads to the generation of an OS command.

Thus, the evidence used for determination of the constituent fragments might derive from a process as simple as referring to a taxonomy of tokens to a process as complex as complete control and data flow analysis of the subject program.

The evidence for the fragment set being of minimal size depends upon many factors. Again, expert judgment could be elicited, but testing a wide variety of programs and sampling the OS command contents of the programs when operated in a benign environment might provide a more compelling body of evidence.

9.4.4 False Positive Failures

A compelling false negative argument could be constructed for a security defense that identified all OS commands as malicious. Every malicious command would be detected, but the rates of false positive failures would be unacceptable. Thus, the S^3 defense attempts to determine the difference between malicious and benign OS commands. The false negative argument discussed above provides the rationale for belief that attacks are detected. Here we discuss the argument that benign OS commands are not identified as attacks.

Similarities to the False Negative Argument

In developing the false positive argument, we use the same initial strategy that we used for the false negative argument, i.e., argue over all relevant OS command locations that false positives failures are reduced ALARP.

For each relevant OS command location, false positives are demonstrated to be reduced ALARP using the same general goals that were used in the false negative argument:

1. Command Interception Adequacy,
2. Command Parsing Correctness
3. Fragment Set Adequacy, and
4. Detection Algorithm Correctness.

Each goal within the false positive argument is used to balance the corresponding goal within the false negative argument. For example, to reduce false negatives, each relevant OS command location must be intercepted. To reduce false positives, we must demonstrate that only relevant OS command locations are intercepted when an OS command is actually executed.

Most of the false positive goals could be supported by evidence in the form of expert judgment. For example, experts could conclude that command interposition is not known to intercept functions spontaneously when the function is not actually called. We note that such evidence for false positives does not rely on the location of the OS command as heavily as false negatives do. Many claims might therefore be argued generally over all locations in a single argument.

Fragment Sources and Fragment Alteration

An important difference between the false positive and the false negative arguments is that the false positive argument depends heavily on characteristics of the particular piece of software that is modified, especially within the fragment set completeness sub-argument. For the fragment set to be

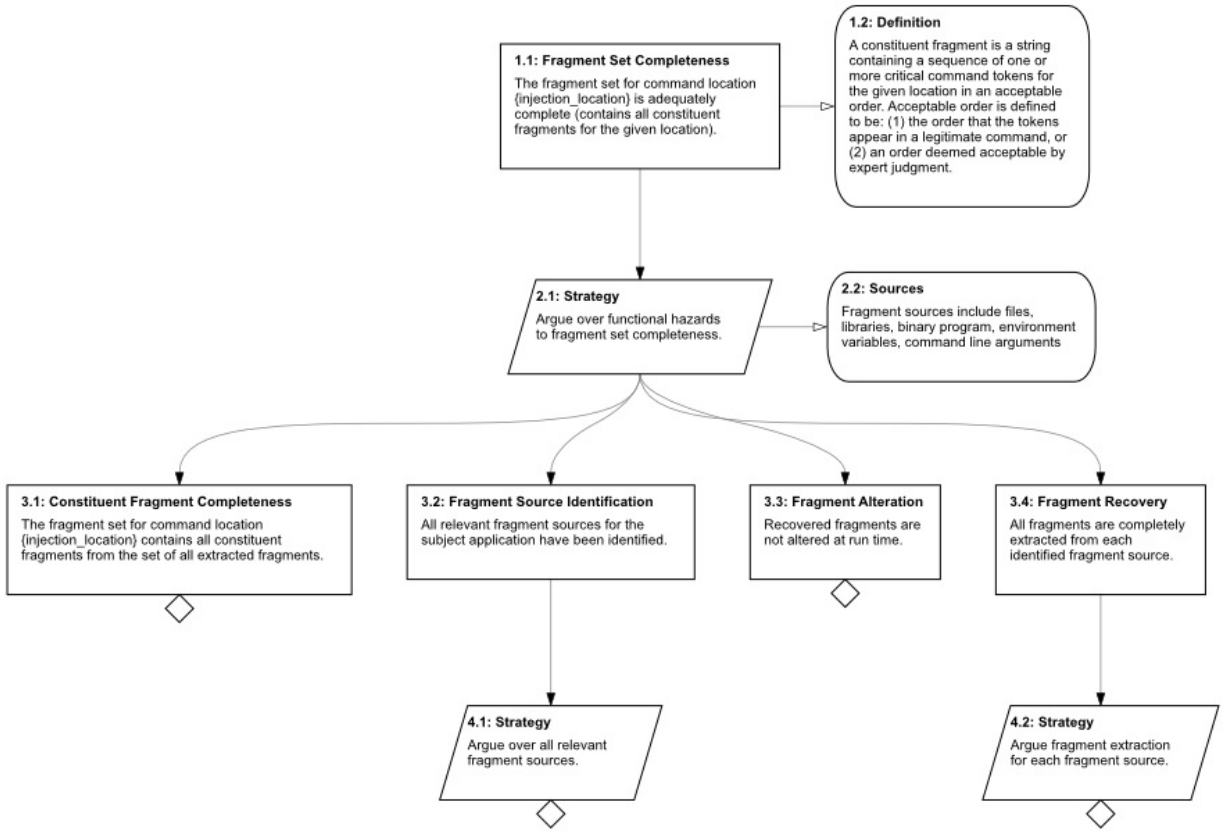


Figure 9.7: Fragment set completeness sub-argument summary

complete (see Figure 9.7), all possible sources of fragments must be identified, and then fragments must be adequately extracted from each source.

The present prototype implementation of S^3 relies on the source of fragments being the binary program and relevant libraries. There is an assumption that fragments either do not originate from other locations or can be easily extracted from other sources. For the purposes of building a strong argument, we cannot rely on this assumption. Instead, we must demonstrate how application of S^3 accounts for the possibility of fragments in alternative locations. In principle, fragments might exist in any of the following:

- The binary program.
- Libraries references by the binary program.

- Files used for configuration of the programs OS commands.
- Environment variables.
- Command line arguments.

The argument must therefore show that all fragment sources have been identified and properly considered in the argument. Each of the above sources is an example of a fragment source that could be overlooked easily.

The success of identifying fragment sources and the success of fragment extraction are therefore dependent on the specific characteristics of the subject application. In our S^3 assurance case, we include a goal to that effect but note here that the associated evidence would probably be limited to control and data flow analysis of the binary program to detect possible input sources, expert judgment, or some kind of evolving taxonomy.

The prototype implementation of S^3 also relies on the assumption that fragments, once extracted, are never altered. Since fragment extraction occurs only once, if the fragments were altered after extraction, all fragment sets might be invalidated. Some users might be willing to accept the assumption that fragment alteration does not occur, but a generalized fragment set completeness argument (shown in Figure 9.7) requires a goal to this effect together with the associated sub-argument. Evidence in support of this goal might include data flow analysis of fragments, instruction analysis of string manipulation operations, or expert judgment.

Constituent Fragment Set Completeness

The final goal in support of fragment set completeness demonstrates that of all extracted fragments, the constituent fragment set for the specific OS command location is complete. In the prototype implementation of S^3, only one fragment set was developed for the entire program. The fragment set was considered complete if all fragments were extracted. For our argument, however, since fragment sets might differ for each OS command issue location, we must reevaluate the concept of a complete fragment set.

To produce a complete fragment set, we could also use all extracted fragments as the fragment set for each OS command location; however, such a policy conflicts with the false negative argument. In the false negative argument, our goal was to narrow the fragment set as much as possible. Specifically, we required that a fragment set at an OS command issue location must contain "constituent" fragments (defined in Section 9.4.3). Based on the concept of a constituent fragment, for a given fragment set to be complete, the argument must:

1. show that all possible fragments have been extracted (discussed above) and
2. show that a given fragment set has all the fragments that are also constituent fragments (a subset of all extracted fragments).

As with the false negative argument, evidence used for determination of the constituent fragments could be derived from a process as simple as referring to a taxonomy of tokens or a process as complex as complete control and data flow analysis of the subject program.

9.5 Addressing Argument Incompleteness

One challenge in developing the S^3 assurance case was that the argument does not mirror the prototype implementation of S^3. Instead, the argument was developed to be compelling generally. As a result, areas of the argument are undeveloped because S^3 would require further alteration and study before the argument can be completed. Argument development continued in an abstract sense by reasoning about:

1. the general form of a completed S^3 argument,
2. the kinds of issues likely to be encountered, and
3. the kinds of data that will be required.

Even when reasoning about the argument hypothetically, development of the argument was fundamentally limited. Some crucial information necessary to complete the argument is simply

unavailable until S^3 is applied to a specific software system. To allow the assurance case to be completed, missing evidence was hypothesized to be available. The assurance case was then repaired using selection argumentation to express the need to validate the missing data.

This section describes several example arguments taken from the S^3 assurance case. For each example, an argument was found to be based on evidence that cannot be provided during development of the argument because the evidence is based on characteristics of a specific software system. For each example, we describe the form of the data hypothesized to be available, and the use of selection argumentation. Section 9.6 summarizes the discovered results from this activity.

9.5.1 The Command Location Confidence Argument

Recall that our approach to arguing false positives and false negatives was to argue over each OS command location. For example, Context 2.2 in Figure 9.5 references all OS command locations for a given piece of software. The locations are not known ahead of time, hence, to complete the argument, a confidence argument is necessary to demonstrate that all OS command locations have actually been identified. Failure to identify any locations would invalidate the false negative argument.

A static analysis of the software would be necessary to find all OS command locations. Therefore, the confidence in Context 2.2 is largely based the given static analysis. The confidence argument would argue that hazards that could negatively affect the static analysis are mitigated.Regardless of which analysis is used, identification of OS command locations could fail for the following reasons:

- No relevant OS command locations are identified.
- The software is obfuscated beyond what the static analysis can handle.
- The development resources necessary to locate OS command locations and to determine if the software is obfuscated exceeds a defined threshold.

All of the above failure conditions depend on the characteristics of a given software system. To complete the confidence argument for Context 2.2 would require hypothesizing that all the above

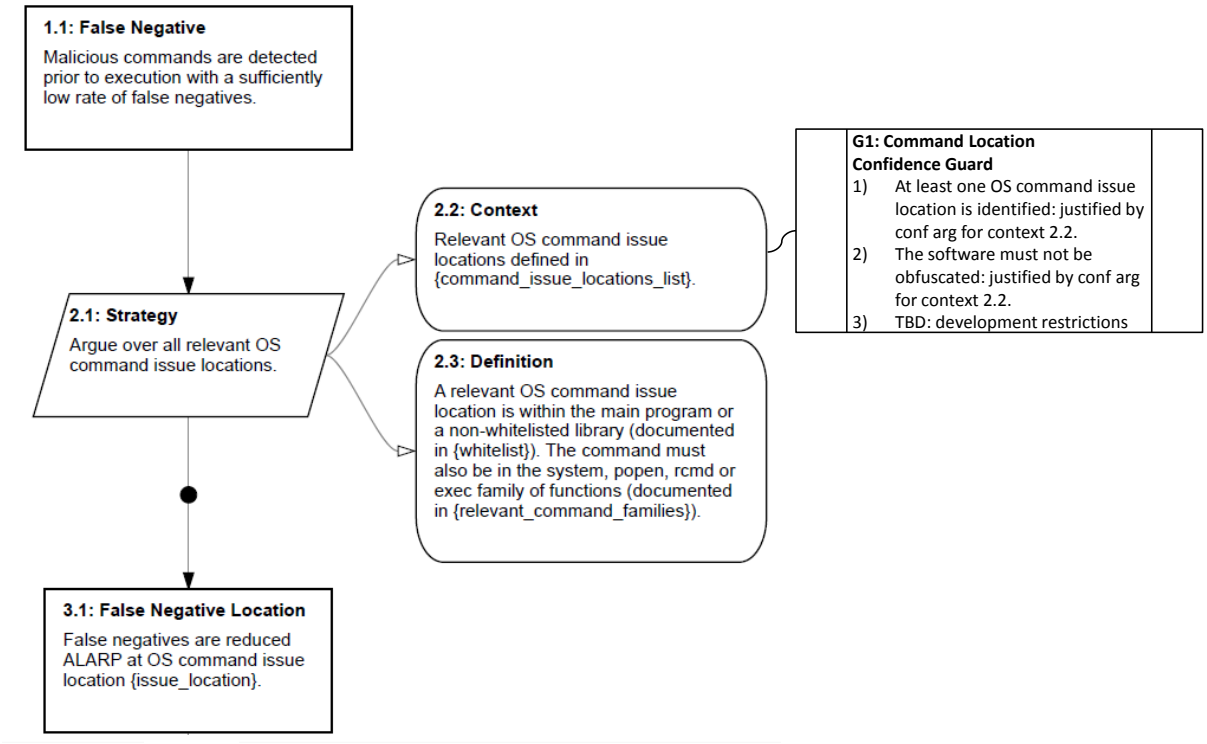


Figure 9.8: Prototype guard for Context 2.2

hazards are mitigated, but each hypothesis cannot be validated until the modification is applied. All of the above hypotheses restrict Context 2.2; hence, we placed a guard on Context 2.2 summarizing the hypothesized data as constraints that will require validation (see Figure 9.8)[4].

Alternative arguments to address invalidated guards are not explored in this study; however, if no relevant OS command locations are identified and the software is not obfuscated, an immediately apparent alternative might be to argue the security threat is adequately mitigated without the application of S^3 .

[4]For the example, the exact constraints for determining if the software is obfuscated are not provided. Additionally, developmental constraints are not specified, but left for the system stakeholders to make a decision about what constraints are appropriate.

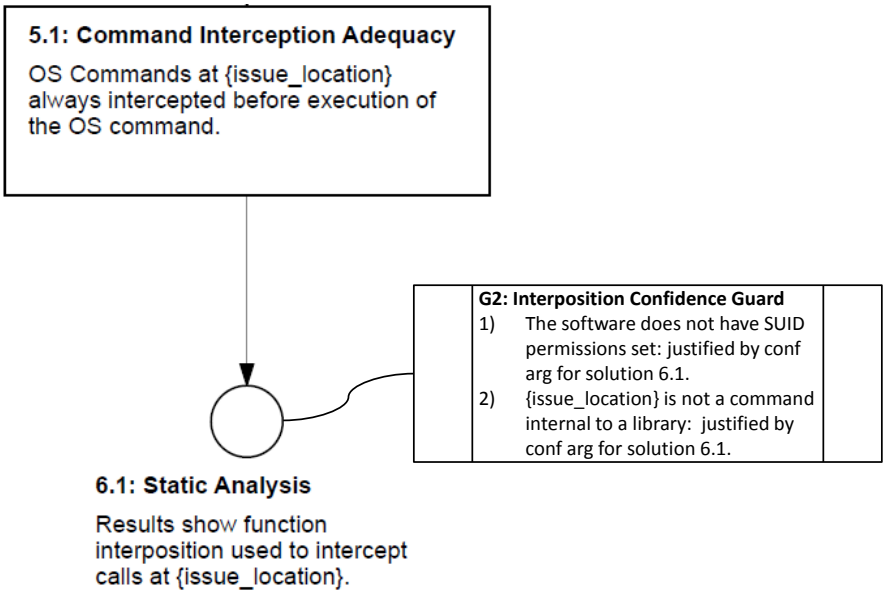


Figure 9.9: Prototype guard for Solution 6.1

9.5.2 The Function Interposition Confidence Argument

In Figure 9.5, the solution element 6.1 uses function interposition as evidence to support that commands issued at a given OS command location are intercepted. By mandating that attack detection differs for each OS command location, function interposition (as originally used in the prototype implementation of S^3) might no longer be applicable as a solution. For example, the address of the OS command might be a necessary parameter for attack detection, but function interposition does not add additional parameters to function calls. For simplicity of the example, we assume the S^3 engineers can find a way to continue to use function interposition and also identify the precise OS command location. Instead, we focus on other applicability concerns of function interposition.

A confidence argument for the solution element 6.1 in Figure 9.5 must demonstrate that hazards to the applicability of function interposition are mitigated. Example hazards include:

- If the software has SUID permissions, traditional function interposition (using LD_PRELOAD) will not work.

- If the software has a copy internally of an OS command library, function interposition is not possible.

When constructing the confidence argument, we cannot assume that these hazards are mitigated (i.e., that the hazard is non-existent). Instead, we hypothesize the hazards are mitigated to complete the confidence argument. A guard is placed to summarize the hypothesized data. In this instance, the guard is placed on solution element 6.6 (see Figure 9.9).

9.5.3 The Remediation Correctness Argument

When an attack is detected by the S^3 approach, there are three prescribed options for attack remediation:

- Terminate execution of the software.
- Repair the command by removing malicious components from the command string.
- Return an error code specific to the OS command type.

A potential risk with remediation is that the behavior after the attack is remediated might violate the stakeholders' concept of correctness.

For example, returning an error code would stop a malicious OS command from being executed. Continued execution of the software, however, might be considered too divergent from expected behavior since continued execution might be based on successful execution of the command. Depending on how the stakeholders define correctness of execution, the same remediation policy might not be applicable for all OS command locations.

If an appropriate remediation policy is dependent on the location of the OS command in the program, then the remediation argument must argue over all individual OS command locations. Further, a selection is required to determine the best remediation policy for each location, illustrated in Figure 9.10.

While excluded from the figure for simplicity, since the strategy of argument is based on each OS command location, Strategy 2.1 would require the same contexts and guards as described above

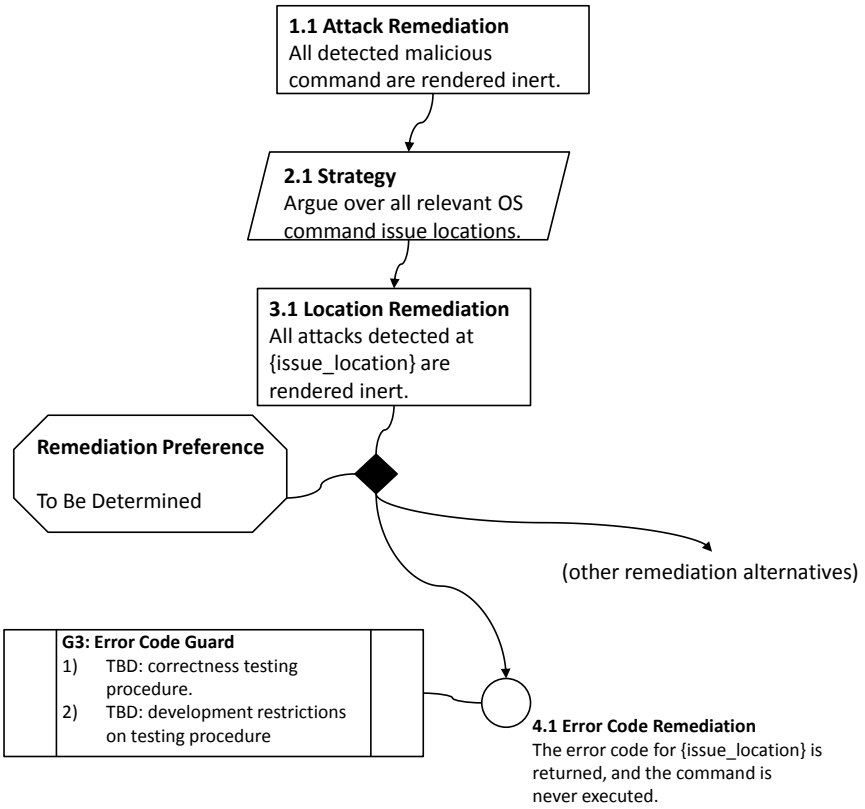


Figure 9.10: Remediation argument prototype

in Section 9.5.1. Because of space limitations, the figure only highlights the remediation policy of returning an error code specifically. Since preference between remediation policies is based on the needs of the system stakeholders, no particular decision model is specified to select between remediation policies. Instead, the decision model indicates that a choice is available among the alternatives and stakeholders will have to supplement the decision model later.

Evaluating correctness in this scenario requires an evaluation of the characteristics of a given instance of software. For example, stakeholders might require testing to demonstrate remediation will result in acceptable execution behavior. A correctness argument based on testing would require the following evidence:

- Demonstration that all tests successfully “pass”, i.e., do not unacceptably change program behavior (correctness argument evidence).

- Demonstration that test inputs are available and/or easily generated (success argument evidence).
- Demonstration that the test inputs provide sufficient "coverage" for the testing procedure (confidence argument evidence).
- Demonstration that testing is completed within a reasonable amount time (success argument evidence).

Static analyses might also provide support for a correctness argument; however, the results of all analyses would be based on a subject instance of software and the data is not known a prori nor guaranteed to be available in the future.

Correctness must first be validated to choose an appropriate remediation policy, which is not known when developing the argument. Each remediation policy is therefore guarded by the above evidence that would demonstrate that the remediation policy is adequately correct. Figure 9.10 provides a stubbed out guard based on testing, but the guard cannot be completed until a testing procedure is defined (this is a topic for future work).

9.5.4 The File Identification Argument

To reduce false positives, an argument is necessary to demonstrate all sources containing fragments have been identified (under Strategy 4.1 in Figure 9.7). To argue that all file sources have been identified, a file identification goal was added under Strategy 4.1 stating:

"All file fragment sources have been identified."

Further development of the file identification argument was not possible because the S^3 prototype does not provide any justification for how to deal with fragments from files. We instead examined the general methods that could be used to justify the file identification argument.

There are essentially two general approaches to file identification:

1. Intercept all file open operations at run time and identify files containing fragments during execution (i.e., *dynamic file identification*).

Approach	Benefits	Consequences
Dynamic File Identification	All files that are opened will be examined.	Increased complexity of the S^3 implementation and the assurance case
	Fragment sets will not contain fragments from files unless the file is opened.	Increased run-time overhead.
Static File Identification	No increased complexity in the S^3 implementation and the assurance case.	Fragment sets will contain fragments from files even if the file is not opened.
	No increased run-time overhead.	Static analysis increase development resource consumption and might fail to find all file sources.

Table 9.2: Benefits and consequences of dynamic and static file identification

2. Identify all files containing fragments prior to execution the software (i.e., *static file identification*).

The ultimate choice between a dynamic or static approach to file identification is at the discretion of the system stakeholders. The benefits and consequences of both approaches are summarized in Table 9.2. The implications of both approaches in terms of argument development are described in the following subsections.

Dynamic File Identification

The fundamental limitation of a dynamic approach is that it will drastically increase the complexity of the S^3 implementation, the assurance case, and the run-time overhead of the modified software, for the following reasons:

- All file open operations will be analyzed to (1) determine if the file contains fragments, (2) extract fragments if necessary, (3) determine what fragment sets should be updated, and (4) assess the quality of altered fragment sets.

- Dynamically updating fragment sets can have broader affects throughout the assurance case, requiring a complete review and revision of the assurance case.

In terms of assurance case development, the file identification argument would be based on successful interception and analysis of all file open operations. The primary limitation to assurance case development is the potential for unacceptable increases in run-time overhead of the modified software system.

Analyses could be performed on a statistically relevant set of programs to show that generally dynamic file identification is acceptably efficient; however, assessments about a given application of S^3 might still be necessary. Example assessments include:

- running the modified software against a serious of test inputs (similar to that described in Section 9.5.3), or
- static analysis of the expected frequency of file interception.

The results of efficiency assessments such as these are not known when developing the efficiency argument. We would therefore have to hypothesize that the efficiency results are acceptable to complete the assurance case. The file identification argument would be guarded based on efficiency assessments.

Further, the manner in which file open operations are intercepted might also require guarding constraints, similar to the constraints described above for OS command interposition (see Section 9.5.2).

Static File Identification

While dynamic file identification relies on one fundamental approach, i.e., interposition of file open operations, static file identification approaches can be much more diverse. Example static file identification strategies are summarized in Table 9.3.

For each static file identification approach in Table 9.3, uncertainties and limitations are provided. Without a specific software system, we cannot determine that these limitations are properly addressed; hence, for each of these methods development of the argument would require hypothesizing that each limitation has been addressed. While the table only presents a partial list for

Identification Strategy	Summary	Uncertainty/Limitation
Demonstrate the software does not open files.	The simplest approach is determine that the software does not have file open operations, thereby obviating the need to extract fragments from file sources.	Most applications likely open files.
Extract all file names from the software.	Analyze all strings in the software and all file open operations. Use data flow analysis to find the name/location of all opened files.	The software could be obfuscated or sufficiently complex such that there unacceptable doubt that the analysis is complete.
Identify files based on heuristics.	Assume that files are found in fixed locations and/or with fixed file extensions.	This approach depends on assumptions about the software behavior and the operating context.
Identify files based on expert judgment.	Experts examine the software and determine what files (if any) the software uses that contain fragments.	Experts might be unavailable or might not have sufficient expertise about a given instance of software.

Table 9.3: Example static file identification techniques

illustration, we anticipate that any approach to static file identification will have similar failure limitations/uncertainties. Any hypothesized data can be expressed as a guard restricting the file identification argument.

Further, static file identification is performed when the modification is generated. There might be a significant risk of using too many development resources. Hypotheses about development resources consumed might also be necessary and therefore might also be expressed within a guard.

9.5.5 The Fragment Extraction Confidence Argument

Once file fragment sources are identified (either dynamically or statically), all fragments must then be extracted from each file. To argue successful fragment extraction, a goal is added under Strategy 4.2 in Figure 9.7 stating:

"All file fragments in file fragment sources are extracted."

This goal is ultimately supported by evidence about the efficacy of a fragment extraction algorithm. The prototype implementation of S^3 does not specify a method for extracting fragments from files. However, regardless of which extraction algorithm is eventually used, the algorithm will likely depend on an assumption about file format. For example, a file could contain one string containing thousands of characters. OS command fragments might be found within a substring, but the fragment extraction algorithm would need to know how to isolate fragments.

The applicability of the extraction algorithm would likely be argued in a confidence argument. In that argument, evidence must be provided that all files have the expected format. Since the file formats cannot be guaranteed when developing the argument, argument development requires a hypothesis that the file format has been shown to be acceptable.

We therefore anticipate the need for a guard for the fragment extraction goal (under Strategy 4.2), where the guarding constraint specifies the format of each file must be shown to have.

9.5.6 Fragment Set Arguments

Fundamentally, any goal stating characteristics of a fragment set will likely be supported by data that cannot be determined until the fragment sets have been computed. In the prototype implementation of S^3, the quality of fragment sets was not derived based on each OS command location, nor was the quality/characteristic of acceptable fragment sets rigorously assessed. Goals based on the fragment set qualities will require further research and development in terms of the data that will adequately support these goals.

The fragment alteration goal in Figure 9.7 will likely be supported by an analysis demonstrating a specific instance of software does not alter fragments once they are extracted. The analysis could perform data-flow analysis on fragments or look for idioms suggesting alteration has occurred. Regardless of what the analyses are used, the fragment alteration argument cannot be developed until the analysis is performed.

Similarly, the fragment set constituency and minimization goals, shown in Figures 9.6 and 9.7, are also dependent on the characteristics of each fragment set. Constituency could be justified based

on data-flow analyses and expert judgment. Minimization might also be determined by data-flow analyses and heuristics.

While further development and study of S^3 is required to better reason about how these goals might be supported, we anticipate all of these goals will be based on data that is hypothesized to be available. We therefore anticipate the need for guards on all of the above goals to express all hypothesized data as constraints on the validity of each sub-argument.

9.6 SSM Results

This section presents the results of our examination of assurance case development as a method to assess and demonstrate the applicability of SSM. These results are summarized in Table 9.4, and described in further detail in each subsection.

9.6.1 SSM Applicability: Identifying SSM Hypotheses

The fundamental purpose of SSM is to validate iteratively *hypotheses* about modifications (see Chapter 3). For an assurance case to reveal the applicability of SSM, the characteristics of SSM hypotheses within the assurance case must be defined. A fundamental result of this study was deriving this definition.

Development of some arguments within the assurance case was not possible unless we *hypothesized* that necessary data would be available and of the proper form when the modification is applied. Hypothesized data represents fundamental uncertainties associated with a modification that directly affect assurance claims within the assurance case. Examination of this approach to argument development yielded the following results:

- An SSM hypothesis is defined as an argument that is predicated on the availability data that is hypothesized, but not guaranteed to be present.
- The hypothesis assessment activities of SSM correspond to validating the hypothesized data is indeed available.

Result	Summary	Section Containing Further Details
The definition of SSM hypotheses.	Discovery and definition of the fundamental characteristics of the assurance case that demonstrate the applicability of SSM (i.e., arguments based on hypothesized data).	Section 9.6.1
Argument guards specify SSM.	Discovery that the use of argument guards provide the basis for specifying SSM within the argument. Further SSM/argument development can use guards as a driver.	Section 9.6.2
The definition of SSM uncertainty.	Discovery and clarification of the uncertainty SSM is meant to address, i.e., uncertainty about necessary but unavailable information.	Section 9.6.3
The distinction between assumptions and SSM hypotheses.	Discovery that GSN assumptions and SSM hypotheses are distinct and the distinction is based on the desires of the system stakeholders.	Section 9.6.4
SSM can be made artificially applicable.	Discovery that SSM could be applicable in all cases if stakeholders demand extreme restrictions on resources used in modification generation.	Section 9.6.5

Table 9.4: Development results summary

Data is hypothesized to be available if:

1. evidence necessary to support a fitness claim is not known until a modification is applied, or
2. evidence necessary to support a fitness claim might not be generated within developmental constraints specified within the success argument.

In the former case, hypothesized data corresponds to missing evidence within the fitness argument. In the latter case, evidence is not necessarily missing within either the fitness or success arguments; however, there is a hypothesis that an instantiation of the SSM process will be able to generate necessary evidence within stakeholder-defined development constraints. For example, stakeholders could terminate any analysis that takes longer than 10 minutes to execute. Within the success argument, a claim is made that all analyses terminate within 10 minutes, which is justified with

evidence about the instantiation of the SSM process. If an analysis necessary to generate evidence for a fitness claim takes longer than 10 minutes to execute, the data will not be available (i.e., the analysis will be prematurely terminated). Hence, all arguments that rely on the results of analyses that are subject to premature termination are based on an underlying hypothesis that each analysis will complete within 10 minutes.

9.6.2 The Principle of Specifying SSM: Argument Guards

Identifying SSM hypotheses (i.e., arguments based on hypothesized data) indicates the potential applicability of SSM. To apply SSM, the argument must express SSM process mechanics through the use of selection argumentation. We discovered during this case study that argument guards are fundamental in specifying SSM process mechanics.

Recall that selection argumentation expresses a selection process to derive an assurance case instance, consisting of three fundamental concepts:

1. product line argumentation,
2. argument guards, and
3. decision models.

Decision models combined with product line argumentation can be used to specify complex selection processes; however, a selection process does not necessarily imply that the process is *speculative* (i.e., that the process is an SSM process). The fundamental difference between a selection process and speculation is that speculation includes the possibility of making a bad selection.

Argument guards express constraints requiring further assessment and indicate situations that might invalidate a selection. Arguments based on hypothesized data (i.e., SSM hypotheses) might exist throughout the fitness argument. Guards summarize hypothesized data as constraints restricting the selection of key components of the fitness argument. Since the fundamental purpose of S^3 is to improve security, all guards in the S^3 were expressed as restrictions to components of the security requirements argument (see Chapter 6 for a more detailed discussion of this approach).

When a guard is invalidated, the argument that is guarded is invalidated, i.e., the guarded argument cannot be used to support an assurance claim. This property of guards implies a selection process *even if no other alternatives are specified*. This observation leads to the following key results:

- Guards are the fundamental concept of selection argumentation for specifying a speculative selection process (i.e., SSM).
- By reviewing guards, developers can determine how to proceed in SSM development, i.e., by finding alternative arguments in case a guard is invalidated. Alternatives can then be included through the use of product line argumentation and decision models.

9.6.3 Differentiating Uncertainty from SSM Hypotheses

Assurance cases are defeasible, i.e., there is always uncertainty about the argument and software system the argument supports. While the purpose of SSM is to reduce uncertainties about software modification, not all uncertainties are reduced by SSM. Based on the above definition of SSM hypotheses, we were able to clarify the uncertainty SSM is meant to address. Specifically:

SSM addresses uncertainties about the existence
of data necessary to support an assurance claim.

Some data might be uncertain or even unavailable prior to applying a software modification, but unless the data directly affects an assurance claim, SSM is not applicable.

For example, structural abstraction is used in the S^3 argument to argue over every OS command location. Entity abstraction is used to reference OS command location identifiers within argument goals (denoted as $\{issue\_location\}$). The OS command location identifier, while uncertain when the argument is developed, can be generated by using the address of each OS command location in memory. There is no question that the identifier can be generated, just uncertainty about the exact form of the identifier. This uncertainty is therefore not an SSM hypothesis because it does not affect the validity of any assurance claims.

9.6.4 Assumptions vs. SSM Hypotheses

This study has revealed an overlap between the concept of an assumption and our definition of an SSM hypothesis.

An assurance case can be based on assumptions. If stakeholders determine an assumption is appropriate, the assumption can be expressed in a GSN assumption element (see Chapter 2). Generally assumptions are considered acceptable as long as there is sufficient confidence (see Chapter 5) that the assumption will always hold. In many cases, the development of the assurance case in this study highlighted assumptions made by the S^3 implementation. For example, the S^3 prototype relies on the assumption that the software being modified is not obfuscated and the assumption that fragments do not originate from files.

Stakeholders must determine whether or not an assumption requires validation. If validation is required, the properties of the assumption are essentially data hypothesized to be available. The argument based on the hypothesized data is therefore an SSM hypothesis. If, however, sufficient confidence exists that an assumption will always be valid, then a GSN assumption element is specified in the argument.

Fundamentally, distinguishing between SSM hypotheses and argument assumptions is at the discretion of the stakeholders.

9.6.5 Artificial Applicability

Another result of this study was the discovery that SSM hypotheses can always be identified if the system stakeholders place extreme restrictions on the development resources used to generate software modifications. The applicability of SSM could therefore be demonstrated *artificially* if we were to demand such restrictions.

For example, analyses necessary to produce a modification can be restricted by the amount of time allowed for the analyses to execute. These restrictions are expressed within the success argument. Even if there is proof that the analyses will not exceed some upper bound, stakeholders could always demand greater restrictions than can be proven. Any argument that relies on the results

of these analyses is consequently identified as an SSM hypotheses: the argument is based on the hypothesis that the analysis time will be within a defined threshold of acceptability.

To avoid artificially demonstrating the applicability of SSM, we did not place specific developmental restrictions on the S^3 modification. Instead, we noted developmental restrictions that appear non-trivial (based on our judgment) and might reasonably qualify as an SSM hypothesis.

9.7 S^3 Results

Building an assurance case for any system requires a detailed understanding/analysis of the system in question, which can often expose important observations/results. These results can be used to further guide system development and research. Consequently, while the primary intent of this study is to evaluate the SSM approach (see Section 9.6), some results can also be reported about S^3 itself.

The study yielded the following general results about S^3:

- The argument affirmed the importance of known issues originally reported by the S^3 authors [84], such as:
  - the need for more advanced data flow analyses to prune fragment sets,
  - the risk of fragments originating from sources outside of the binary program, potentially increasing false positive attack detection,
  - the risk of binary obfuscation thwarting fragment recovery algorithms,
  - the risk of alteration of strings at run time thwarting fragment recovery based on static analysis, and
  - the risk of fragment extraction recovering too many fragments leading to increased false negative attack detection.
- The argument required a detailed analysis of the design minutia of S^3 suggesting alterations to the design and the need for further research:

- The argument exposed elements of the design of S^3 for which the use ad hoc methods of analysis are questionable.

- The argument forced the consideration of all assumptions (either explicitly stated or implicit) of the approach.

The original S^3 prototype and publication were intended to demonstrate a proof of concept and motivate continued research. Consequently, the original prototype relied on reasonable assumptions for the purpose of demonstration, and the scope of the original evaluation is necessarily narrow and preliminary. By no means is this a criticism of their work; however, there is a distinction between demonstrating a proof of concept and demonstrating high assurance of production software. Development of an assurance case is by its nature focused on developing high assurance about critical properties of a system. Narrow results and seemingly reasonable assumptions must be questioned and justified to produce a high-quality assurance argument.

Below is an itemized list of specific key results about S^3 discovered as a result of developing the assurance case:

- The efficacy of S^3 string recovery relies on two ad hoc recovery algorithms: one that recovers strings that are found statically in the binary, and another approach that recovers strings that are assembled by the program during execution. The argument revealed the need for additional evidence about the efficacy of these algorithms to justify the completeness of fragment sets.

- A formal specification of the algorithm used to parse strings, especially to parse formatted strings, would be beneficial in bolstering assurance claims about the completeness of fragment sets.

- All analyses used by S^3 should have explicitly defined failure semantics. The argument revealed that freedom of analysis warnings was found to be necessary evidence to justify confidence that fragment sets are adequately complete and correct.

- The argument revealed the need for additional analyses (either automatic or manual) to justify assumptions used for the initial prototype. Specifically, analyses are necessary to determine if the given program is obfuscated or if the program processes strings abnormally during execution. Obfuscation or abnormal string manipulation can affect the efficacy of fragment recovery, yet there is no justification that these issues can be ignored or assumed away.

- More evidence is necessary about the role of S^3 in the context of other similar attack classes. Attackers might be able to inject OS commands that are already blessed. Further, attackers might be able to manipulate actual blessed fragments without any form of injection through the program semantics. While these issues are somewhat out of scope for S^3, the argument revealed that these larger concerns must be considered to determine if S^3 will be generally effective.

- The published S^3 results cannot be reasonably used with an assurance case. Confidence concerns (i.e., are the results complete, appropriate, and correct) could not be resolved with the current data. Future evaluation of S^3 should be engineered to provide evidence in support of specific assurance claims, thereby producing a strong argument for practical deployment of software modified by S^3.

- The argument revealed that a single attack remediation policy for all injection locations might unnecessarily increase performance overhead or potentially result in unexpected program behavior. In some cases, a desired policy might not be applicable for a specific location (e.g., returning an error code is only valid if the error code is known). Remediation policies should be customized per injection location. Additional analyses are therefore necessary to providence evidence in support of assurance claims that the appropriate remediation was chosen for each injection location.

- The argument revealed that the possibility of fragments originating outside of the program is a more serious concern than originally proposed. Configuration files, environment variables and command line arguments might reasonably have legitimate (i.e., blessed) fragments. Further

support for fragment recovery from these sources is crucial to support assurance goals that false positive rates are adequately minimized[5].

- Verification of the implementation, specifically verification of the algorithms for fragment parsing, fragment extraction, critical token identification and attack detection would provide valuable evidence supporting assurance goals related to false positive and false negative attack detection.

- The argument forced further evaluation of the efficacy of command interposition. Command interposition was found to not be applicable for all injection locations. Further evidence of the efficacy of command interposition for S^3 is necessary to justify assurance claims that no relevant injection locations are missed.

- The argument revealed that false negatives and false positives can be further reduced by customizing fragment sets per injection location. Producing an argument for this technique was difficult because of the use of command interposition. Command interposition appears to be better suited when using a single attack detection and remediation policy for all injection locations. Consequently, command interposition might not be the best option of OS command interception for S^3. Other approaches should be researched and considered.

- The argument revealed the need for a precise description of the characteristics of "ideal" fragment sets (i.e., fragment sets that *adequately* minimize false negatives and false positives). For example, fragment sets might be ideal if they contain a number of fragments under a certain threshold or fragments of a certain type, etc. The argument revealed that justifying assurance claims about acceptable false positives and false negatives relies on knowing the fragment sets have these yet to be determined ideal characteristics.

- False positive detection negatively impacts correct program behavior, but the argument revealed it is not the only cause of incorrect program behavior. Remediation policies might

[5]Subsequent discussions with the S^3 developers revealed they were aware of this issue and had already partially addressed the issue in the latest prototype

negatively affect correctness depending on the semantics of individual injection locations. For example, returning an error code might send subsequent execution down undesirable execution paths. While the impact of remediation on correctness is implied by the argument, establishing the different ways remediation can negatively impact correct behavior requires additional research and new analyses.

9.8 Case Study Results Summary

This case study evaluated SSM to determine the practicality of applying SSM. Specifically, this case study evaluated the development of an assurance case to answer the question "Can SSM be applied to other modification approaches and how?" (see Section 9.1). The results of this evaluation are summarized below.

9.8.1 SSM Applicability for S^3

This case study has revealed the potential applicability of SSM to another modification approach (i.e., S^3). The term "potential" is used because the applicability of SSM ultimately depends on the needs of specific stakeholders (which is subjective) and also depends on further development of S^3 (which is out of scope for this study). This case study has highlighted practical concerns of applying S^3 (see Section 9.5), i.e., concerns that fundamentally affect the acceptability of software systems modified using S^3. Practical application of S^3 cannot ignore these concerns, but simultaneously, many issues cannot be resolved until S^3 is applied.

We have proposed how to address these scenarios through the development and validation of hypotheses about data that will be generated. Selecting and validating hypotheses describes an SSM process. This case study has demonstrated how the argument can be built both to reveal and express hypotheses within the argument structure, i.e., through the use of guards. This study has therefore exposed concerns about the application of S^3 that can be resolved through the use of SSM.

While determining the applicability of SSM is a stakeholder decision, the results of this study suggest many concerns within S^3 that could be resolved with SSM. Practical application of S^3 would have to address these concerns in some manner; hence, we conclude that the applicability of SSM for S^3 is likely.

9.8.2 Argument Development Defines SSM Applicability

The novelty and importance of this case study is the demonstration of how SSM can be applied from first principles. The primary focus was to provide a general approach and guidelines engineers can adopt to apply SSM for their own purposes. The primary result of this study is that the development of an assurance case can be used to assess the applicability of SSM and to apply SSM concepts. Development of an assurance case therefore facilitates repeatability of SSM and consequently development of an assurance case supports the practical application of the SSM model.

Through development of the assurance case, hypotheses requiring assessment can be identified. Hypotheses are identified as arguments based on evidence that cannot be known during the development of the assurance case. Argument development continues by hypothesizing that the data is available. By applying selection argumentation (specifically guards), the dependency of arguments on hypotheses is highlighted and the mechanics for validating hypotheses are specified. Further review of argument guards can drive further development of the argument, and further development of more complex decision processes.

This case study was limited to an initial iteration of assurance case development; however, development of the argument has exposed concerns requiring further research and development in terms of the S^3 technology, the assurance case and acceptable alternatives for addressing invalidated guards. We anticipate that further development iterations will expose further potential applications of SSM. The results of this study therefore suggest that development of an assurance case can serve as the guiding principle by which SSM is practically applied.

Chapter 10

Related Work and Current Practices

10.1 Software Product Line Engineering

As discussed in Chapter 6, there are some similarities between SSM and the principles of software product line engineering. Software product line engineering is general engineering paradigm based on the concept of customization of software products through the reuse of a set of existing *software platforms* [62]. The goal is to provide software solutions that are tailored as much as possible to the needs of any particular consumer, but to develop the software using already developed software subsystems and interfaces (i.e., software platforms).

The key motivations for software product lines are to [62]:

- support increased demand by consumers for software customization,
- reduce complexity of supporting increased customization demands,
- reduce development costs and time to market through maximum reuse of available software platforms,
- enhance software quality by using software platforms that have been widely reviewed and tested, and
- improve development cost and effort estimates.

Software product line engineering consists of two fundamental activities:

Domain Engineering: The commonality of and variability the product line can offer are defined and realized.

Application Engineering: A specific software system is constructed using the reusable components defined during domain engineering.

SSM shares some similarities with software product line engineering concepts. Specifically, SSM involves selecting a method to modify software from potentially many alternatives. This activity is similar to product line application engineering, and involves similar selection activities: i.e., assessments are performed to determine what selection is best based on the predefined alternatives.

A fundamental difference, however, is that SSM prescribes variability and selection to account for unknowns/uncertainties involved with modifying software. Software product lines use variability and selection fundamentally to make maximum reuse of available components to decrease engineering costs and effort. SSM can be thought of as *reactive* where software product line are *proactive*. That is, selection and variability in SSM is a consequence of uncertainty and not ideally desired, whereas in software product lines, variability is desired and the product line is built to maximize reuse of software artifacts. Another key difference is that SSM bases selection and validation of alternatives based on an assurance case. The assurance case is therefore the driver in SSM. Previous work on product line assurance cases serve primarily as a supporting document [22].

Despite the key differences, the similarities of SSM and software product line engineering might in the future be shown to provide mutual benefits. For example, our assurance based approach of finding uncertainties within an assurance case (see Chapter 9), and using SSM assessments to validate these uncertainties might be a beneficial approach for domain engineering in software product lines. Similarly, the SSM process of finding an acceptable modification might be beneficial to and benefit from the software product line concept of application engineering. We leave further investigation into the concept of *assurance based product line engineering* for future work.

10.2 Assurance Based Development

Many of the concepts used in SSM take advantage of concepts originally proposed in Assurance Based Development (ABD) [23, 30, 31]. ABD is a software development paradigm prescribing the development of a software system in parallel to the development of an assurance case. The key concept of ABD is that assurance goals should be considered early and often in the development of software. In this manner, assurance goals drive development activities, and the end result of development is a software system that is considered adequately fit for use as demonstrated in the co-developed assurance case.

ABD is in contrast to previous approaches where assurance cases are developed after a software system is implemented. Developing an assurance case after the development of the software is problematic. After the software has been developed, it is difficult to recover exactly how the implementation realizes an assurance goal. Further, even if an assurance case is recovered, it might indicate the software system is not fit for use (e.g., the argument might be unconvincing or highlight assurance concerns that were never addressed). Altering and re-engineering a software system after primary development is often costly and inefficient.

For a software system to be fit for use, it must provide adequate functionality within any stakeholder-defined operational constraints (e.g., efficiency). In ABD, development of the software is predicated on the development of a fit-for-use (i.e., fitness) argument. The development of the fitness argument is limited by development constraints, i.e., costs and schedule restrictions. ABD accommodates developmental constraints by mandating the construction of a separate success argument. The success argument justifies that the development activities will result in a software system that is fit for use within the defined cost and schedule constraints.

The development of both the success and fitness arguments drive the development of the software system. This symbiotic development of the assurance case and the software involves two primary activities, illustrated in Figure 10.1:

Process synthesis: Unsupported goals, referred to as *assurance obligations*, within the fitness and

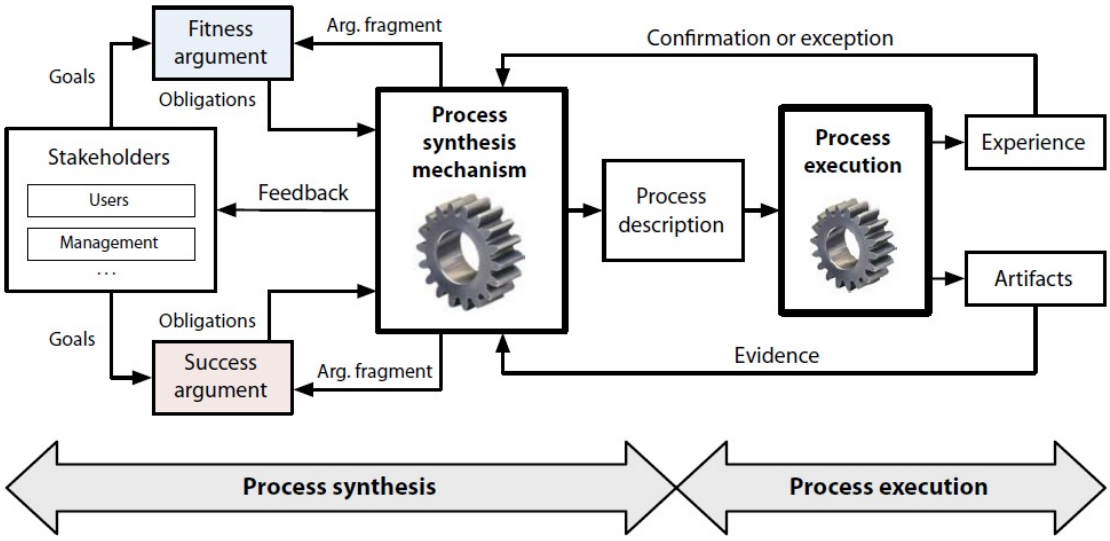


Figure 10.1: The ABD approach – figure taken from Graydon [23]

success argument are used to develop a *process description* also referred to as a *planned process*. The process description defines a set of activities to perform in order to support assurance obligations.

Process Execution: The activities specified in the process description are then performed, i.e., executed. Process execution can either be successful, generating evidence necessary to support assurance obligations, or unsuccessful, in which case a new process description must be synthesized.

Process synthesis and process execution are iterative activities. Process synthesis and process execution continues until a complete fitness assurance case (and consequently a completed software system) is constructed. In each round of process synthesis, illustrated in Figure 10.2, the developer:

1. selects one or more assurance obligations to address
2. assembles a set of candidate options to address the selected assurance obligations
3. assesses each candidate option and then selects one
4. updates the planned process (process description) to reflect the activities associated with the selection option.

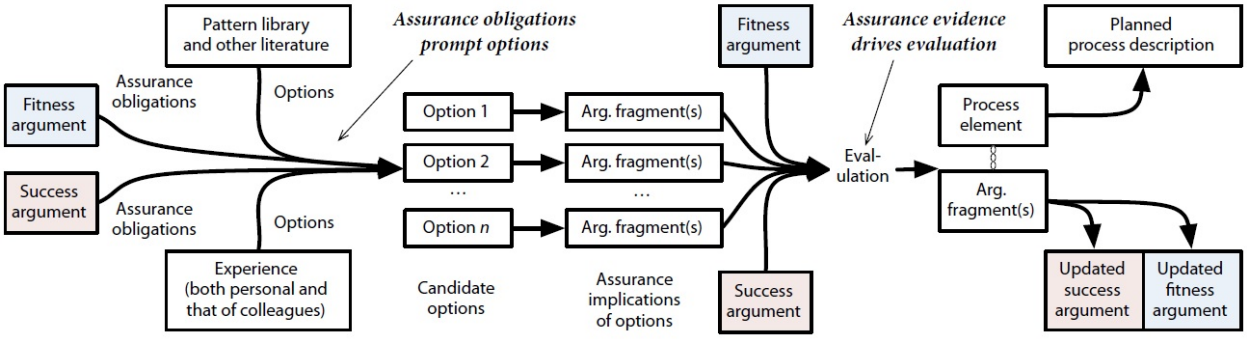


Figure 10.2: ABD process synthesis — figure taken Graydon [23]

5. updates the fitness and success arguments to reflect expected evidence that will be generated by the new planned process.

When executing the planned process, developers might find the process fails to address assurance obligations adequately. In this case, the planned process must be altered, and by extension, the associated arguments must be repaired. ABD defines a repair mechanism to address the possibility of an unsuccessful planned process. Conceptually, the ABD argument repair mechanism involves:

- removing elements of the success and fitness arguments that are associated with approaches that failed to address any given assurance obligations, and
- assuring the removal of argument elements leaves the argument in a consistent state, i.e., the repaired argument no longer makes reference to removed elements of the argument.

10.2.1 Comparison of ABD and SSM

SSM is in essence an extension of the ABD concept. In SSM, the development of software modifications by an instantiated SSM process is predicated on assurance arguments in the SSM assurance case. The interdependence of the SSM process on the assurance case makes an SSM an ABD-like process. ABD and SSM therefore share many similarities. Specifically, both ABD and SSM:

- define a development process

- prescribe the use of success and fitness arguments
- take into consideration multiple options by which assurance goals can be met
- prescribe mechanisms by which alternative options are selected if found to be invalid

Despite the similarities, SSM and ABD are distinct, and have several differences:

- The SSM development process is predefined and can it itself be instantiated using any software development paradigm. ABD, however, is an engineering paradigm to develop a single software system.
- The ABD success argument is moot and disregarded once engineering of the software system is complete. In SSM, the success argument is an artifact that must be generated and supplied to the stakeholders, much like the SSM fitness argument and instantiated SSM process.
- SSM allows for argument construction when evidence is missing using the argumentation mechanics discussed in Chapter 6. In ABD, if evidence cannot be generated, the approach is excised from the argument using the ABD argument repair mechanism.

The key difference is that SSM is a predefined process and can itself be the insantiated using any software development paradigm, including ABD. This characteristic allows us to draw the distinction between the general SSM model and an SSM instantiation. The SSM model is a development process, but developers instantiate the model to define a reusable process to modify any relevant SOUP. In Chapter 9, we adopted a development methodology for instantiating SSM based on an ABD-like process in which development of the assurance case drives development of SSM.

10.3 Assurance Driven Design

Assurance Driven Design (ADD) [86] is an extensions to the Problem-Oriented Software Engineering (POSE) framework [87]. POSE is a framework for engineering design, based on three basic elements:

Problem statements: a description of a problem engineers must solve which describes the contexts, requirements, and solutions.

Problem transformations: a series of transformations used to convert problems into problems that are easier to solve.

Justifications: Each transformation is accompanied by a justification that guards the problem transformation. The transformation must be shown to be adequate (based on the justification).

POSE involves a similar cyclic exploration and validation structure as prescribed by both ABD and the SSM process model. The key difference is that POSE is not guided by an assurance case. Also, POSE has been primarily used to find and validate system designs and not complete software systems including their implementation and other development concerns.

In ADD, the justifications of accepted POSE transformations are used as an argument that the solution solves the intended problem. Because justifications are reviewed and validated, ADD justifications are also similar in concept to SSM guards. The difference is that in ADD, justifications are used to construct an argument, and selection argumentation guards are restrictions to existing GSN arguments. Also, SSM guards are explicit about the characteristics that must be validated. ADD justifications are based on an unspecified assessment performed by *problem-owning* and *solution-owning* stakeholders.

10.4 Justifying the Use of SOUP

Adelard (a dependability consulting agency) has developed a comprehensive report for justifying the use of SOUP in safety-critical software systems [11]. In their report, they propose an approach

for the justification of safe SOUP by linking safety justification activities and evidence to the safety life cycle specified in the standard IEC 61508 [88]. The report provides general overview of the issue with SOUP is safety critical systems, and the general kinds of evidence that might be used to support/justify various dependability characteristics. This report could serve as a starting point to determine what kinds of evidence would adequately justify an assurance claim in an SSM assurance case.

10.5 The Common Criteria

The Common Criteria is a security standard whereby general security requirements for a class of similar products are specified in *protection profiles* [89]. A software user can specify a protection profile in order to locate acceptable software solutions, or software vendors can specify protection profiles in order to make claims about the security offered by their products. For example, a user might require that all software used in their organization meets all the requirements specified in a defined protection profile. A vendor must demonstrate that their software conforms to the requirements specified by the protection profile before use of the software is permitted.

A *target of evaluation* (TOE) is a specific piece of software that is evaluated with respect to a protection profile. A *security target* documents how the TOE conforms to the requirements of a given protection profile. For example, a security target documents the security-critical assets and threats to those assets and the *security objectives*, i.e., the countermeasures to defend against defined threats. Third parties then evaluate the protection profiles and security targets. These evaluations are used as evidence in support that the TOE has the desired security properties.

Conceptually, protection profiles and security targets form an implicit assurance argument, i.e., if the TOE is validated to have the specified properties, then the TOE is secure. Assessment of the validity of this implicit argument is at the discretion of the stakeholders and not defined within the common criteria.

Adopting the Common Criteria to SSM might be possible and beneficial; however, a full assurance case is still necessary to provide a comprehensive assurance claim. The systematic and explicit decomposition of the argument from a top-level goal forces stakeholders to consider all relevant threats to security that could otherwise be overlooked. The Common Criteria might therefore be applicable within the SSM model but as a supplement to (not a replacement of) the SSM assurance case.

10.6 Contracts for Modularity and Software Reuse

Argument guards used by selection argumentation share many characteristics in common with the general concept of *contracts* that are used to enable modularity in developing software systems [90, 91, 92]. In this use of the term, a contract specifies general characteristics that are required for a specific component of a software system. Developers either implement a solution or find a reusable component that satisfies the contract. In this respect, a contract and an argument guard serve the same general purpose, i.e., a solution must be found that satisfies or validates its contract or guard.

Contracts have also been applied within assurance cases. To simplify the development and management of assurance cases, modularity extensions to GSN arguments have been proposed [93, 19]. Large and complex arguments are grouped into modules yet some interdependencies remain. For example, a goal in a module might be supported by an argument found in another module. Rather than copying the supporting argument fragment (which would undermine the purpose of modularizing the argument), interrelationships are expressed using GSN contract elements.

GSN contracts specify how a goal is supported by arguments and evidence found in other modules. Argument guards also express interrelationships within the argument, but the interrelationship is with respect to selection restrictions. The restrictions on selection may or may not have any relationship to supporting a given goal. Argument guards are therefore fundamentally distinct from GSN contracts as they serve a completely different purpose.

10.7 Search-based Software Engineering

For each stage of the software engineering life cycle, engineers must generate solutions that balance competing constraints and, in doing so, engineers must choose between many potential solutions. The field of Search-based Software Engineering (SBSE) [94, 95, 96, 97] is based on the observation that most software engineering problems are essentially optimization problems. By reformulating software engineering problems as search-based problems, search-based optimization algorithms (i.e., metaheuristics) can be applied [98], such as simulated annealing [99], genetic algorithms [100], and tabu search [101], to find optimal solutions from a potentially large solution space.

A key principle in formulating problems as search-based problems is the definition of a *fitness function* [98]. A fitness function characterizes a "good solution", and is applied to solutions to impose an ordinal scale for solution comparison. While the SSM model does involve a search process, SSM does not necessarily use a traditional fitness function and therefore is not necessarily described in terms of SBSE. Selecting between alternatives in SSM fundamentally involves selection between alternative arguments. An argument is considered "fit for use" if all guarding constraints are valid. This concept of fitness does not prescribe an ordinal comparison between solutions as is necessary for SBSE.

In SSM, preference between alternatives are represented using decision and refinement models. If stakeholders do not have a preconceived notion of preference between alternatives, these decision could be represented using search-based decision models. In this case, a selected argument must demonstrate that all guarding constraints are valid *and* that the selected argument represents the most optimal solution as defined by the fitness function. The construction of an assurance case for SSM, however, lends itself to the development of a predefined preference between alternatives. That is, engineers would likely choose the most desirable alternative initially when constructing an SSM argument and only specify alternatives when the possibility of failure becomes evident (as described in the case study in Chapter 9).

Chapter 11

Conclusion

This dissertation has defined and evaluated *Speculative Software Modification*. SSM prescribes an iterative process of selecting and validating hypotheses about how to modify a specimen of SOUP. By altering SOUP to improve dependability, SSM facilitates the practical use of SOUP in dependability-critical operating environments. SSM also allows for the practical use of software modification technologies by explicitly assessing/validating potential hazards affecting the acceptability of generated modifications. Assessments are based on uncertainties about applying a SOUP modification and the needs of the system stakeholders, both of which are expressed within the assurance case. If an assessment fails to validate a method for modifying the SOUP, SSM prescribes an iterative selection process based on the assessment results. A key novelty of SSM is the iterative selection process is defined in and driven by an assurance case.

This work presents a number of contributions:

- The SSM model and mechanics, defined in Chapter 3, a novel assurance-based approach for practical SOUP modifications under uncertainty to improve software dependability.

- The general SSM assurance case form, consisting of a fitness and success argument, which supports SSM process mechanics and is the fundamental driver of all SSM activities, defined in Chapter 3.

- The fitness argument for security development methodology and general structure, defined in Chapter 4. This structure facilitates structured and rigorous reasoning about the acceptability of security-enhancing SOUP modifications.

- The argument-based security metric framework, described in Chapter 5, a novel framework for measuring software security based on the assessment of confidence in an assurance argument.

- Selection argumentation notation and mechanics, which allow an assurance case to support the SSM process mechanics, defined in Chapter 6.

- Argument guards, a novel component of selection argumentation that express argument selection restrictions based on assessments that must be performed.

- Stack Layout Transformation, a novel approach for protecting binary programs against buffer overflow attacks, defined in Chapter 7.

- An assurance driven approach for applying SSM from first principles, described and explored in Chapter 9. This approach provides developers a practical method to determine the applicability of SSM and engineer SSM incrementally.

- Two case study assessments of the benefits of assurance cases for software modifications for security described in Chapters 8 and 9. These case studies suggest that assurance cases for security are crucial and should be further studied and applied.

- Case study assessments of SSM through the examination of the SSM process mechanics, the SSM assurance case, and the applicability of the SSM model, described in Chapters 7, 8, and 9. These studies indicate the potential feasibility and practicality of SSM.

This dissertation presents a body of work that supports the conclusions described in the following sections.

11.1 Arguments are Essential for Practical Software Modification

A recurring result from this dissertation is that arguments are crucial to the practical application of software modifications, regardless of the applicability of SSM. In this dissertation, two specimen modification approaches were analyzed through the development of assurance cases (see Chapters 8 and 9). In both instances, unsupported goals were found extensively through the argument. These goals were essential to supporting the top-level assurance claim, yet minimal or no evidence was available to support these claims.

While in some cases missing evidence was a result of uncertainty about the characteristics of the software being modified, in many cases, goals were unsupported because of a lack of knowledge about the modification approach in general. Development of an assurance case therefore exposes fundamental limitations in SOUP modification approaches requiring further research and development.

Without an explicit assurance argument, hazards that affect modification assurance negatively are easily overlooked. The current practice of evaluating software modification techniques in an ad hoc manner has a tendency to obscure the claims that are supported by any available evidence. Additionally, some evaluations tend to focus on very specific characteristics of a software modification, like run-time efficiency, whereas other practical concerns affecting overall acceptability are minimally addressed.

While ad hoc evaluations and implicit arguments might provide a proof of concept or initial evidence of the utility of a modification technique that is appropriate for academic publication, to apply these approaches in practice requires a comprehensive and explicit argument.

This dissertation has demonstrated that:

Development of an assurance case forces a systematic exploration of all hazards affecting modification acceptability, thus promoting the practical application of software modifications. .

11.2 SSM is Feasible and Practical

The main thesis of this work is:

> **Thesis:** *Speculative Software Modification is a feasible approach to produce practical SOUP modifications for establishing desired dependability properties in software.*

This work provides evidence from three case studies that suggests the thesis is true:

Process Model Results: Chapter 7 evaluated the feasibility and practicality of the SSM process model described in Chapter 3. This study suggested that the process model can be feasibly instantiated and provide practical benefits. Additionally, the study suggested the need for a more rigorous and comprehensive model of acceptability.

SSM Assurance Results: Chapter 8 evaluated the feasibility and practicality of applying assurance cases to the SSM concept. This study demonstrated the benefits of assurance cases for SSM but also exposed fundamental limitations with traditional assurance case techniques in support of SSM. As a result, selection argumentation was invented and applied. Combining selection argumentation with an assurance case was shown to feasibly and practically facilitate documentation of an explicit and comprehensive acceptability model for SSM. By providing a method for modeling SSM assurance, the SSM assurance case further supports the practicality of SSM.

Applicability Results: Chapter 9 addressed the feasibility and practicality of SSM in terms of the applicability of the SSM model. We examined how engineers could determine if SSM is applicable from "*first principles*", i.e., how can engineers determine if SSM is applicable and develop the SSM model. The development of an assurance case was used as the driver for determining the applicability of SSM for a specimen modification approach. The results of this study demonstrate a method in which engineers can systematically and iteratively develop SSM. Further, this study provides evidence of the applicability of SSM for a second modification approach.

The conclusion of the feasibility and practicality of SSM is limited by the scope of each case study. Ideally, a statistically valid set of specimen SSM implementations would be evaluated across a range of modification techniques and dependability properties. In practice, fully implementing and evaluating SSM would involve cooperative development and review from stakeholders, software developers and assurance case developers.

Each case study was primarily conducted and evaluated by the author; hence, evaluation of SSM was limited to two specimen modification approaches and to the three above targeted evaluations of SSM. Each study was therefore limited in the extent to which SSM processes and assurance cases were developed (described in each case study chapter). Additionally, since the author both conducted and evaluated each case study, there is a potential for bias in any conclusions about SSM. While any doubt about conclusion bias cannot be completely removed, each case study was conducted systematically to the best of our effort to record both positive and negative results.

Despite the limitations of the evaluations of SSM, the results provide initial evidence suggesting SSM is both feasible and practical. As an initial evaluation, we anticipate that further evaluations of SSM will result in further SSM model refinements and more detailed examinations of the benefits and applicability of SSM.

11.3 Future Work

This work has exposed many potential directions for future SSM research as well as research into related topics:

The SSM Development Life Cycle: A more detailed SSM development approach has yet to be explored. Questions still remain about (1) how engineers determine the adequacy of their SSM implementation and SSM assurance case, (2) when SSM engineering should terminate in specifying alternatives, (3) the specific mechanics for using an SSM assurance case as a specification for implementation, and (4) how the SSM model is updated and altered (i.e., maintained) post deployment.

The SSM Deployment Life Cycle: The results of each case study suggest that SSM might exist within a larger SSM deployment cycle. SSM would therefore be one module in a pipeline of activities. For example, prior to executing an SSM process, stakeholders might first select the specific kind of acceptability model that is appropriate. Additionally, after SSM terminates, further assessment of produced modifications and the assurance case instance might be necessary.

Selection Argumentation Tool Support: In this dissertation, SSM assurance cases were developed with minimal tool support. Currently, no assurance case tools provide any support for selection argumentation. To support practical application of SSM, assurance case tools should be created that both display selection argumentation and also organize and manage the potentially complex interactions and processes described in selection argumentation.

Complex Selection Processes: The mechanics and use of selection argumentation were purposefully simplified for this dissertation to decrease the complexity in introducing and evaluating the concept (described in Chapter 6). Future work could further explore how selection argumentation can be used to handle arbitrarily complex selection processes.

Modification Assurance Case Templates: This work has suggested that research into software modification approaches should always be based on the development of a rigorous argument. A potential direction for future work would be how researchers can develop assurance case templates/prototypes for their modification technologies. The original developers and researchers should have a much deeper knowledge of their technology and are therefore best suited for developing the initial assurance case. Then, when practitioners apply these approaches, they are not burdened with developing the argument independently. Instead, practitioners can focus on the finer details of how to adapt the technology for their use.

Software Quality Metrics Based on Argument Confidence: In Chapter 5, a security metric framework is proposed based on argument confidence. In this dissertation, the concept was included as a potential direction for examining the quality of arguments but further evaluation of the

approach is necessary. Another potential direction for future work is the use of confidence assessments in SSM for making selections between alternatives and more formal definitions of the acceptable solution space in terms of a *confidence space*.

Assurance-Based Software Product Line Engineering: The overlap of SSM and software product line engineering requires more evaluation and study. A potential novel direction for future work suggested by SSM is the development of product lines based on development of an assurance case. Such an approach would be a generalization of SSM and of Assurance Based Development (described in Chapter 10).

Bibliography

[1] Philip L Carret. *The art of speculation*. John Wiley & Sons, 1997.

[2] Merriam-Webster. Speculation, 2014.

[3] Oxford Dictionaries. Speculation, 2014.

[4] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[5] John Viega and Gary McGraw. *Building secure software: how to avoid security problems the right way*. Pearson Education, 2002.

[6] Gregory Wallace. Michaels stores: Possible data 'attack'. http://money.cnn.com/2014/01/25/news/companies/michaels-security-breach/, January 2014.

[7] Hayley Tsukayama. Michaels discloses possible customer data break; secret service investigating. http://www.washingtonpost.com/business/technology/michaels-discloses-possible-customer-data-breach-secret-service-investigating/2014/01/27/73a8538e-877c-11e3-a5bd-844629433ba3_story.html, January 2014.

[8] Dennis Lynch. Michaels, target, and more: The biggest and most sensitive data breaches in recent history. http://www.ibtimes.com/michaels-target-more-biggest-most-sensitive-data-breaches-recent-history-1548033l, January 2014.

[9] T. Winograd, H.L. McKinley, L. Oh, M. Colon, T. McGibbon, E. Fedchak, R. Vienneau, Information Assurance Technology Analysis Center (IATAC)., Data, and Analysis Center for Sofware (DACS). *Software Security Assurance: A State-of-the Art Report (SOAR)*. Information Assurance Technology Analysis Center, 2007.

[10] Felix Redmill. The cots debate in perspective. In Udo Voges, editor, *Computer Safety, Reliability and Security*, volume 2187 of *Lecture Notes in Computer Science*, pages 119–129. Springer Berlin Heidelberg, 2001.

[11] P.G. Bishop, R.E. Bloomfield, and P.K.D. Froome. *Justifying the Use of Software of Uncertain Pedigree (SOUP) in Safety-related Applications*. HSE contract research report. University of Southampton, Institute of Sound and Vibration Research, 2001.

[12] Gogul Balakrishnan and Thomas Reps. DIVINE: Discovering variables in executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse Engineering (RE)*, pages 45 – 54, 2002.

[14] Penny Grubb and Armstrong Takang. *Software Maintenance: Conepts and Practice*. World Scientific Publishing Company, 2003.

[15] Felix Redmill. *ALARP Explored.* Technical report series. University of Newcastle Upon Tyne, Computing Science, 2010.

[16] Timothy Patrick Kelly. *Arguing safety: a systematic approach to managing safety cases*. University of York, 1999.

[17] Ministry Of Defence Standard. Standard 00-56 issue 4-safety management requirements for defence systems. *UK Ministry of Defence*, 2007.

[18] T. Kelly and R. Weaver. The goal structuring notation–a safety argument notation. In *Proc. DSN 2004 Workshop on Assurance Cases*, 2004.

[19] Goal Structuring Notation Working Group et al. Gsn community standard version 1, 2011.

[20] RE Bloomfield, PG Bishop, CCM Jones, and PKD Froome. Ascadadelard safety case development manual, 1998.

[21] Peter Bishop and Robin Bloomfield. A methodology for safety case development. In *Industrial Perspectives of Safety-critical Systems*, pages 194–203. Springer, 1998.

[22] Ibrahim Habli and Tim Kelly. A safety case approach to assuring configurable architectures of safety-critical product lines. In *Architecting Critical Systems*, pages 142–160. Springer, 2010.

[23] Patrick John Graydon. *Assurance Based Development*. PhD thesis, University of Virginia, 2010.

[24] John Rushby. Mechanized support for assurance case argumentation. In *Proceedings of the 1st International Workshop on Argument for Agreement and Assurance. Springer*, 2013.

[25] Patrick Graydon, John Knight, and Mitchell Green. Certification and safety cases. In *The 28th International System Safety Conference*, Sept. 2010.

[26] Benjamin D Rodes and John C Knight. Speculative software modification and its use in securing soup. In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 210–221. IEEE, 2014.

[27] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[28] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[29] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[30] Patrick J. Graydon, John C. Knight, and Elisabeth A. Strunk. Assurance based development of critical systems. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 347–357, Washington, DC, USA, 2007. IEEE Computer Society.

[31] P.J. Graydon and J.C. Knight. Software process synthesis in assurance based development of dependable systems. In *Dependable Computing Conference (EDCC), 2010 European*, pages 75–84, 2010.

[32] Richard Hawkins, Tim Kelly, John Knight, and Patrick Graydon. A new approach to creating clear safety arguments. In *Advances in Systems Safety*, pages 3–23. Springer, 2011.

[33] Rob Alexander, Richard Hawkins, and Tim Kelly. Security assurance cases: motivation and the state of the art. Technical report, CESG/TR/2011, 2011.

[34] John Goodenough, Howard Lipson, and Chuck Weinstock. Arguing security-creating security assurance cases. *rapport en ligne (initiative build security-in du US CERT), Université Carnegie Mellon*, 2007.

[35] Howard Lipson and Chuck Weinstock. Evidence of assurance: Laying the foundation for a credible security case. *rapport en ligne (initiative build security-in du US CERT), Université Carnegie Mellon*, 2008.

[36] R. Bloomfield, S. Guerra, M. Masera, A. Miller, and Saydjari O. Workshop report for the workshop on assurance cases for security. http://www.csr.city.ac.uk/AssuranceCases/Assurance_Case_WG_Report_180106_v10.pdf, 2005.

[37] Samantha Lautieri, David Cooper, and David Jackson. Safsec: Commonalities between safety and security assurance. In *Constituents of Modern System-safety Thinking*, pages 65–75. Springer, 2005.

[38] B Dobbing and S Lautieri. Safsec: Integration of safety and security–safsec methodology: Standard, 2006.

[39] Gary Stoneburner. Sp 800-33. underlying technical models for information technology security. 2001.

[40] Michael Whitman and Herbert Mattord. *Principles of information security*. Cengage Learning, 2011.

[41] Bruce Schneier. Attack trees. *Dr. Dobbs journal*, 24(12):21–29, 1999.

[42] Mitre. Common weakness enumeration. http://cwe.mitre.org/.

[43] Greg Hoglund and Gary McGraw. *Exploiting online games: cheating massively distributed systems*. Addison-Wesley Professional, 2007.

[44] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[45] Chris Salter, O Sami Saydjari, Bruce Schneier, and Jim Wallner. Toward a secure system engineering methodolgy. In *Proceedings of the 1998 workshop on New security paradigms*, pages 2–10. ACM, 1998.

[46] H.A. Watson. Launch control safety study. *Bell labs*, 1961.

[47] John Knight. *Fundamentals of Dependable Computing for Software Engineers*. CRC Press, 2012.

[48] Tor Erlend Fægri and Svein Hallsteinsen. A software product line reference architecture for security. In *Software Product Lines*, pages 275–326. Springer, 2006.

[49] C.B. Weinstock, J.B. Goodenough, and A.Z. Klein. Measuring assurance case confidence using baconian probabilities. In *1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE)*, 2013.

[50] Anaheed Ayoub, BaekGyu Kim, Insup Lee, and Oleg Sokolsky. A systematic approach to justifying sufficient confidence in software safety arguments. In *Computer Safety, Reliability, and Security*, pages 305–316. Springer, 2012.

[51] John B. Goodenough, Charles B. Weinstock, and Ari Z. Klein. Eliminative induction: A basis for arguing system confidence. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1161–1164, Piscataway, NJ, USA, 2013. IEEE Press.

[52] Benjamin D Rodes, John C Knight, and Kimberly S Wasson. A security metric based on security arguments. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, pages 66–72. ACM, 2014.

[53] Christopher Alberts, Julia Allen, and Robert Stoddard. Integrated measurement and analysis framework for software security. Technical report, DTIC Document, 2010.

[54] Sal Stolfo, S.M. Bellovin, and D. Evans. Measuring security. *Security Privacy, IEEE*, 9(3):60–65, 2011.

[55] S.L. Pfleeger and R.K. Cunningham. Why measuring security is hard. *Security Privacy, IEEE*, 8(4):46–54, 2010.

[56] S.L. Pfleeger. Useful cybersecurity metrics. *IT Professional*, 11(3):38–45, May 2009.

[57] W. Jansen. *Directions in Security Metrics Research*. DIANE Publishing Company, 2010.

[58] Benjain Rodes and John Knight. Reasoning about software security enhancements using security cases. In *The First International Workshop on Assurance for Argument and Agreement (AAA)*, Oct. 2013.

[59] Debra S Herrmann. *Complete guide to security and privacy metrics: measuring regulatory compliance, operational resilience, and ROI*. CRC Press, 2007.

[60] Ewen Denney, Ganesh Pai, and Ibrahim Habli. Towards measurement of confidence in safety cases. In *The International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 380–383. IEEE, 2011.

[61] Paul Clements and Linda Northrop. Software product lines: practices and patterns. 2002.

[62] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering*, volume 10. Springer, 2005.

[63] Vijayan Sugumaran, Sooyong Park, and Kyo C Kang. Software product line engineering. *Communications of the ACM*, 49(12):28–32, 2006.

[64] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[65] CWE/SANS. Top 25 most dangerous software errors, 2011.

[66] Timothy K. Tsai and Navjot Singh. Libsafe: Protecting critical elements of stacks. Technical report, 2001.

[67] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits. Technical report, 2001.

[68] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[69] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 255–270, Berkeley, CA, USA, 2005. USENIX Association.

[70] Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens. ValueGuard: Protection of native applications against data-only buffer overflows. In *Proceedings of the 6th International Conference on Information Systems Security*, ICISS, pages 156–170, Berlin, Heidelberg, 2010. Springer-Verlag.

[71] H. Etoh and K. Yoda. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003.

[72] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, pages 105–120, Berkeley, CA, USA, 2003. USENIX Association.

[73] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan. Preventing overflow attacks by memory randomization. In *Software Reliability Engineering (ISSRE)*, pages 339 –347, nov. 2010.

[74] Jason D. Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson. MEDS: The memory error detection system. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, ESSoS '09, pages 164–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[75] The PAX Team. http://pax.grsecurity.net.

[76] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[77] Benjamin Rodes, Anh Nguyen-Tuong, JohnC. Knight, James Shepherd, JasonD. Hiser, Michele Co, and JackW. Davidson. Diversification of stack layout in binary programs using dynamic binary translation. Technical Report CS-2012-01, University of Virginia, 2012.

[78] BenjaminD. Rodes, Anh Nguyen-Tuong, JasonD. Hiser, JohnC. Knight, Michele Co, and JackW. Davidson. Defense against stack-based attacks using speculative stack layout transformation. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 308–313. Springer Berlin Heidelberg, 2013.

[79] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[80] Asia Slowinska and Herbert Bos. Howard : a dynamic excavator for reverse engineering data structures. *NDSS*, 2011.

[81] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[82] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS. The Internet Society, 2003.

[83] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmarks. http://www.spec.org/osg/cpu2006, 2006.

[84] Anh Nguyen-Tuong, Jason D Hiser, Michele Co, Nathan Kennedy, David Melski, William Ella, David Hyde, Jack W Davidson, and John C Knight. To b or not to b: Blessing os commands with software dna shotgun sequencing. In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 238–249. IEEE, 2014.

[85] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The worlds fastest taint tracker. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2011.

[86] Jon G Hall and Lucia Rapanotti. Assurance-driven design. In *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*, pages 379–388. IEEE, 2008.

[87] Jon G Hall, Lucia Rapanotti, and Michael Jackson. Problem oriented software engineering: A design-theoretic framework for software engineering. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 15–24. IEEE, 2007.

[88] IEC IEC. 61508 functional safety of electrical/electronic/programmable electronic safety-related systems. *International electrotechnical commission*, 1998.

[89] Common criteria common criteria for information technology security evaluation, 2012.

[90] Irfan Sljivo, Barbara Gallina, Jan Carlson, and Hans Hansson. Strong and weak contract formalism for third-party component reuse. In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pages 359–364. IEEE, 2013.

[91] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects*, pages 200–225. Springer, 2008.

[92] Irfan Sljivo, Jan Carlson, Barbara Gallina, and Hans Hansson. Fostering reuse within safety-critical component-based systems through fine-grained contracts. In *International Workshop on Critical Software Component Reusability and Certification across Domains*, 2013.

[93] Jane Fenn, Richard Hawkins, Phil Williams, and Tim Kelly. Safety case composition using contracts-refinements based on feedback from an industrial case study. In *The Safety of Systems*, pages 133–146. Springer, 2007.

[94] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[95] Mark Harman, Edmund Burke, John Clark, and Xin Yao. Dynamic adaptive search based software engineering. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '12, pages 1–8, New York, NY, USA, 2012. ACM.

[96] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, Kings College London, Tech. Rep. TR-09-03*, 2009.

[97] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[98] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, and Sheppard. Reformulating software engineering as a search problem. *IEE Proceedings-software*, 150(3):161–175, 2003.

[99] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, 1984.

[100] Thomas Back. *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996.

[101] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.