Resilient Post-Quantum Software Updates for Critical Infrastructure Systems

by Brian Romansky

B.S. in Electrical Engineering, May 1992, Rensselaer Polytechnic Institute
M.S. in Electrical Engineering, May 1993, Rensselaer Polytechnic Institute
M.S. in Management and Technology, May 1997, Rensselaer Polytechnic Institute

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of the George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

January 9, 2026

Dissertation directed by

Thomas Mazzuchi
Professor of Engineering Management and Systems Engineering & of Decision Sciences

Shahram Sarkani
Professor of Engineering Management and Systems Engineering

The School of Engineering and Applied Science of The George Washington University certifies that Brian Romansky has passed the Final Examination for the degree of Doctor of Philosophy as of November 6, 2025. This is the final and approved form of the dissertation.

Resilient Post-Quantum Software Updates for Critical Infrastructure Systems

Brian Romansky

Dissertation Research Committee:

Thomas Mazzuchi, Professor of Engineering Management and Systems Engineering & of Decision Sciences, Dissertation Director

Shahram Sarkani, Professor of Engineering Management and Systems Engineering, Dissertation Co-Director

Amir Etemadi, Adjunct Professor of Engineering and Applied Science, Committee Member

Muhammad Islam, Professorial Lecturer of Engineering Management and Systems Engineering, Committee Member

© Copyright 2025 by Brian Romansky
All rights reserved

Dedication

I dedicate this dissertation to Tina: my wonderful wife, trusted collaborator, and favorite lab partner. This long journey would not have been possible without your support, patience, and unwavering encouragement. . .-.. --- ...- . -.-- --- ..-

quo deinde ibimus

Acknowledgments

I would like to thank my advisors, Dr. Mazzuchi and Dr. Sarkani for their guidance, encouragement, and support in defining and shaping this research.

To the anonymous reviewers from IEEE Security & Privacy, thank you for your feedback and insightful recommendations. Your comments brought greater clarity to this work.

I also would like to acknowledge a few people who influenced my career and helped me enter this field of research. Monroe Weiant introduced me to the practice of secure systems and encouraged me to take on a leadership role in advancing the field. James Euchner taught me how to think like a researcher while perpetually executing with a focus on meaningful value propositions. Dr. Sherry Shannon-Vanstone gave me the opportunity to engage in advanced security design work that significantly informed this research. They and many other colleagues and co-workers have informed my views on technology research and security in complex systems.

Abstract

Resilient Post-Quantum Software Updates for Critical Infrastructure Systems

Rapid advances in quantum computing technology pose a grave threat to the methods currently used to protect remote software updates against the introduction of malicious code. This threat is particularly challenging for critical infrastructure systems that are frequently in service for many decades. Addressing this threat will require the adoption of new algorithms for digital signatures. While these new algorithms are believed to be secure against future attacks, they introduce design challenges due to a larger size for each signature and additional computational effort required to validate signatures.

The transition to these new algorithms provides an opportunity to also address a long-standing risk due to a hidden single point of failure that exists in the software update workflow for many systems. The development of advanced software update frameworks makes it possible to address several vulnerabilities that have been observed in the software supply chains of modern systems. These frameworks support features such as the use of multiple digital signatures, eliminating a single point of failure and improving software update resilience.

This research examines the combination of The Update Framework (TUF) with the approved set of post-quantum algorithms known as Stateful Hash-Based Signatures (SHBS). A probabilistic model is developed to show the benefit of using multiple signatures for each update in terms of the relative risk when compared to a single signature. Simulation results are introduced to show the impact of using TUF with SHBS in terms of cumulative network bandwidth consumed and total computational resources required to validate update packages. A genetic algorithm optimization technique is also demonstrated as a method for automating the selection of an optimal or near-optimal set of operating parameters for the integrated system. This work shows that the combination of TUF with SHBS is a practical option for a resilient post-quantum software update solution.

Table of Contents

Dedication . . . . . **iv**

Acknowledgments . . . . . **v**

Abstract . . . . . **vi**

List of Figures . . . . . **x**

List of Tables . . . . . **xii**

List of Acroynyms . . . . . **xiii**

1 Introduction **1**

1.1 Research Motivation 1
1.2 Problem Statement 4
1.3 Research Objectives and Hypotheses 5
1.4 Research Contributions 7
1.5 Overview of Dissertation 7

2 Literature Review **9**

2.1 Software Development Environment 10
2.2 Critical Infrastructure Environment 15
2.3 Historical Attacks Related to Software Updates 18
2.4 Attacks Against Critical Infrastructure Systems 21
2.5 Post-Quantum Algorithms 22
2.6 Software Update Frameworks 26
2.7 Multi-Signature Protocols 28
2.7.1 Cryptographic Message Syntax 29
2.7.2 Trust Anchor Management Protocol 29
2.8 Multi-Signature Algorithms 30
2.8.1 Threshold Signatures 31
2.8.2 Hybrid Signatures 31
2.8.3 Composite Signatures 32
2.8.4 Sequential Signatures 32
2.9 Byzantine Fault Tolerance 32
2.10 Chapter Summary 34

3 Methodology **35**

3.1 Impact of Multiple Signatures 35
3.2 Impact of Post-Quantum Algorithms on TUF 37
3.3 Method for Automated Parameter Selection 37
3.4 Chapter Summary 39

4 Risk Model with Multiple Signatures **40**

4.1 Software Distribution Workflow . . . 40
4.2 Client Configuration . . . 41
4.2.1 Client Security States . . . 42
4.2.2 Client State Transitions . . . 43
4.3 The Role of Signers . . . 45
4.3.1 Signer Workflow . . . 45
4.3.2 Signer Threats . . . 46
4.3.3 Signer States . . . 47
4.4 Concatenated Signatures . . . 49
4.5 Validation Policy . . . 51
4.6 Software Vulnerabilities . . . 52
4.7 Attacker Model . . . 53
4.8 Risk Model . . . 53
4.9 Risk Analysis . . . 57
4.10 Unreliable Signers . . . 59
4.11 Bayesian Estimation and Incorporation of Expert Judgment . . . 60
4.12 Benefit-Cost Ratio . . . 62
4.13 Chapter Summary . . . 65

5 TUF with SHBS . . . 66

5.1 Tutorial on SHBS . . . 67
5.1.1 SHBS Benchmark Data . . . 70
5.1.2 State Management . . . 70
5.1.3 Impact of Parameter Selection . . . 71
5.1.4 Number of Signatures . . . 72
5.1.5 Public Key Size . . . 73
5.1.6 Signature Size . . . 73
5.1.7 Computational Cost of Validation . . . 73
5.1.8 Asymmetric Nature of the Update Process . . . 73
5.2 Tutorial on TUF . . . 74
5.2.1 Key-Pair Rollover . . . 78
5.3 Software Update Sequence . . . 78
5.3.1 Planned Updates . . . 79
5.3.2 Critical Updates . . . 80
5.3.3 Contingency Updates . . . 80
5.4 Realistic Update Sequences . . . 81
5.4.1 Ethernet Switch . . . 82
5.4.2 Embedded Operating System . . . 83
5.5 Impact of SHBS on Basic Updates . . . 84
5.6 Relative Cost of Post-Quantum Software Updates . . . 85
5.7 Simulation Model . . . 85
5.8 Simulation Results . . . 89
5.9 Multiple Roots . . . 91
5.10 Chapter Summary . . . 93

6 Automatic Parameter Selection **95**
6.1 Optimization Method 95
6.2 Cost Function 96
6.3 Parameter Selection and Results 96
6.4 Chapter Summary 99
7 Conclusions and Future Research Recommendations **101**
7.1 Conclusions 101
7.2 Limitations 103
7.3 Future Research 104
Bibliography **106**
A XMSS and XMSS^{MT} Benchmark Data **114**
B LMS and HSS Benchmark Data **116**
C ECDSA Benchmark Data **118**
D TUF Simulation Results **119**
E Python Source Code **124**
E.1 Multinomial Calculation 124
E.2 Support Packages 126
E.3 Main Simulator 132
E.4 Genetic Algorithm for Optimization 133

List of Figures

1.1 Software release process with a single signature. 2

1.2 Mind map of problem statement. 5

1.3 Mind map of problem statement (continued). 6

2.1 Outline of literature categories. 9

2.2 DOD DevSecOps software life-cycle (Lam and Chaillan, 2021). 10

2.3 SECDevOps system life-cycle (Mathieson et al., 2021). 11

2.4 DHS list of critical infrastructure segments (United States Department of Homeland Security, 2013). 16

2.5 Seven phases of the intrusion kill chain for Critical Infrastructure (CI) systems (Capano, 2019). 22

3.1 Linkage from research questions to methodologies. 36

4.1 Simplified software distribution model. 41

4.2 State transitions for a client system. 44

5.1 Simplified view of the TUF process flow. 76

5.2 Release workflow integrated with TUF. 77

5.3 TUF simulation high-level description. 86

5.4 TUF simulation sequence diagram. 88

5.5 Device A with 1 Root signer. 91

5.6 Device B with 1 Root signer. 91

5.7 Device A with 5 Root signers. 92

5.8 Device B with 5 Root signers. 92

6.1 Optimization results using PyGAD. 97

6.2 Optimization Results for 1 and 5 roots. 98

6.3 Optimization results compared with fixed algorithm results. 99

List of Tables

1.1 Challenge of security and availability. 3

2.1 Software update stages from ISO/IEC 62443. 17

4.1 Example for $N = 3$ signers. 55

4.2 Comparing results for multiple signers, $N = 3$, 5, and 7. 59

4.3 Comparing results with $N = 5$ and values of Q. 59

4.4 Unreliable signers with a simple majority quorum. 60

5.1 Comparison of ECDSA with XMSS and LMS 72

5.2 Device A (Ethernet Switch) update sequence. 83

5.3 Device B (Bottlerocket OS) update sequence. 84

5.4 SHBS compared to Elliptic Curve Digital Signature Algorithm (ECDSA) for the basic update scenario. 85

5.5 Comparison of results for 1 and 5 Root signers. 93

6.1 Genetic algorithm options used for SHBS parameter selection. 97

A.1 XMSS and XMSSMT benchmark data 115

B.1 LMS and HSS benchmark data 117

C.1 ECDSA benchmark data 118

D.1 TUF simulation results for Device A (part 1 of 2) 120

D.2 TUF simulation results for Device A (part 2 of 2) 121

D.3 TUF simulation results for Device B (part 1 of 2) 122

D.4 TUF simulation results for Device B (part 2 of 2) 123

List of Acroynyms

APT Advanced Persistent Threat 16, 47

BCR Benefit-Cost Ratio 62, 64, 65, 101, 103

BFT Byzantine Fault Tolerance 32, 33, 104

CI Critical Infrastructure x, 1, 4, 7, 15–18, 21–23, 33–35, 37, 40, 47, 79, 101–104

CMS Cryptographic Message Syntax 29

CNSA Commercial National Security Algorithm 3, 23, 103

CODE Cyber Oriented Digital Engineering 16

CPS Cyber Physical Systems 1, 21

CRL Certificate Revocation List 42, 48, 80

CSS Code Signing System 12

CSV Comma Separated Value 86

DHS Department of Homeland Security 15

ECC Elliptic Curve Cryptography 23, 85

ECDSA Elliptic Curve Digital Signature Algorithm xii, 30, 46, 49, 70, 71, 84, 85, 89, 90, 94, 105

HSM Hardware Security Module 3, 59, 69, 70

HSS Hypertree Signature Scheme 25

IETF Internet Engineering Task Force 23, 25, 27, 32

IoT Internet of Things 17, 28

IT Information Technology 13

LMS Leighton-Micali Hash-Based Signatures 23–25, 66, 70, 71, 81, 85

MBSE Model-Based Systems Engineering 11

MBSO Model-Based System Operations 11

NIPP National Infrastructure Protection Plan 15

NIST National Institute of Standards and Technology 11, 16, 24, 66, 72, 102

NSA National Security Agency 3, 4, 10, 23, 25, 34

PEM Privacy Enhanced Mail 29

PKI Public Key Infrastructure 30, 32, 48, 80, 81

RFC Request for Comment 23

RSA Rivest, Shamir, Adleman 23

SBOM Software Bill of Materials 27

SCADA Supervisory Control and Data Acquisition 17

SCMS Security Credential Management System 29, 30

SDL Security Development Lifecycle 13, 14, 47

SEDevOps Systems Engineering DevOps 17

SHBS Stateful Hash-Based Signatures vi, xii, 3, 4, 6, 7, 23, 24, 32, 34, 37–39, 66–73, 76, 81, 85, 86, 89, 90, 92–99, 101–104

SoK Systematization of Knowledge 20, 26

SUIT Software Updates for Internet of Things 5, 25, 27

TAMP Trust Anchor Management Protocol 29, 42

TUF The Update Framework vi, x, 2–7, 27, 28, 31, 32, 37–39, 42, 65–67, 74–78, 81, 85, 86, 88–91, 93–96, 98, 99, 101–105

XMSS eXtended Merkle Signature Scheme 23–25, 66, 70, 71, 81, 85

ZT Zero Trust 16

Chapter 1: Introduction

1.1 Research Motivation

The software update supply chain presents a paradox to cybersecurity. It brings the benefit that deployed systems can be updated with the latest software to overcome known vulnerabilities. It also introduces the threat that an attacker can cause substantial harm if they compromise the secure keys used to sign code; making the software delivery channel a highly attractive target (Massacci et al., 2021).

The situation is particularly challenging for Critical Infrastructure (CI) systems, which are often composed of a multitude of independent embedded controllers that can cause significant physical damage if they are compromised. Classified as Cyber Physical Systems (CPS), these complex systems make it possible for a digital threat to have real-world, kinetic consequences. These threats are made even more challenging by the fact that many digital controllers may remain in use for decades, meaning that systems built today will need to be protected against threats that will emerge years in the future.

A common practice for protecting software updates is to apply a single digital signature to prove the authenticity and integrity of an update package (Lesk, 2014). A valid signature ensures that only authorized updates from a trusted source with access to the private key used to create the signature will be accepted by remote systems. This approach introduces a single point of failure. If the private key used to sign software updates is compromised or if an attacker gains access to the systems that create the signature, malicious code can be introduced (Ladisa et al., 2023a).

Figure 1.1 shows a model of a typical software distribution process. In this model, a secure development process produces new software which is delivered to a system test process. Software that has passed all required tests is then delivered to a process where a digital signature is applied. Here, a private key is used to apply a digital signature to

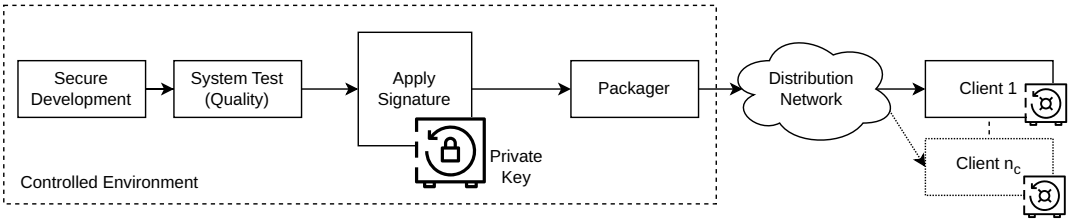

Figure 1.1: Software release process with a single signature.

one or more binary files that correspond to the software that is approved for distribution. Once signed, the software is delivered to a packager that prepares the software update for distribution to one or more client systems. During manufacturing or initial installation, each client system is provided with a trusted copy of the public key that must be used to validate signatures. A detailed description of the digital signature process is provided in Chapters 4 and 5.

Consider the challenge of protecting the privacy of the key used to sign software updates. If an attacker were to gain access to that key, or to the system that has access to the key, then they could sign malicious code which would be accepted by client systems. However, if the private key were to become unavailable due to a system failure, then it would become impossible to deliver future software updates. This creates a conflict, as many of the techniques that are commonly used to make an asset highly available work against efforts to keep it protected against malicious use. Table 1.1 compares several specific techniques to ensure security and availability. In systems where a single private key is used to protect software updates, and client systems are provided with a single public key to validate future updates, this key pair presents a single point of failure.

Prior research into the types of attacks that have been successful against software update systems has motivated the development of new "frameworks" that specify the workflow and structure of a software update package, providing protection against several potential attack types. In particular, Samuel, Mathewson, Cappos, and Dingledine (2010) provide a comprehensive review of attacks against software repositories and introduce The Update

Table 1.1: Challenge of security and availability.

More Secure	**More Available**
Single instance of private key	Multiple copies of private key
HSM with tamper responding circuit to protect private key	Reliable storage for private key
Signing restricted to one system with maximum security protections	Multiple systems available for signing with maximum availability
Small number of users authorized to enable signatures	Many users authorized to enable signatures
Decrease attack surface	Increase redundancy

Framework (TUF), a comprehensive approach that is resilient against key compromise attacks as well as several other attack types. One of the significant attributes of this approach is to use multiple signatures for each role to allow for recovery in response to a situation where a signer key may be compromised. While there is evidence of a slow but successful adoption of TUF, the use of the feature to support multiple signatures is more limited; meaning that systems are not getting the full advantage from the framework.

In addition to this threat, the public key digital signature algorithms that are most frequently used to protect software updates have come under threat from rapid developments in quantum computing (Schneier, 2018). This has motivated a transition from conventional algorithms to new "post-quantum" algorithms that are believed to be resistant to attack from an adversary that has a quantum computer.

The United States National Security Agency (NSA) published the Commercial National Security Algorithm (CNSA) Suite 2.0 which recommends the adoption of Stateful Hash-Based Signatures (SHBS) for systems that require long-term protection (National Security Agency, 2024). While several alternative post-quantum methods have been proposed, SHBS are uniquely valuable because they rely on a limited set of security assumptions (Butin, 2017). However, like other post-quantum options, these algorithms require more computational effort, produce larger signatures, impose a need for state management, and have a complex set of parameter options that are unfamiliar to many developers.

1.2 Problem Statement

The emerging threat from attacks that take advantage of quantum computing technology requires that CI systems must adopt new post-quantum algorithms. At the same time, CI systems are under threat from attacks that attempt to introduce malicious code through the software update supply chain. A practical solution that combines an approved post-quantum algorithm with a mature software-update framework would address both of these challenges.

While there are many options for post-quantum digital signature algorithms, the recommendation from the NSA to use an approved SHBS method is compelling. These algorithms are more mature than many newer alternative signature schemes. The security of these algorithms also relies on fewer assumptions than all other approved post-quantum options. For these reasons, this research will focus on the introduction of approved SHBS for software updates.

Attacks on the software update supply chain take advantage of the hidden single point of failure that is present in many CI systems. The use of a single digital signature to attest to the integrity and authenticity of an update package presents an attractive target for attackers to exploit. TUF was developed to address this specific issue, and it is one of the most mature and extensively deployed update frameworks. This method introduces the use of four distinct roles and multiple digital signatures to protect each update package. It is designed to be effective against many supply chain threats including the compromise of a single signing key. Therefore, this research will adopt TUF as a recommended software update framework.

This research will focus on the combination of SHBS signatures with TUF as a software update solution for CI systems. This novel combination has the potential to address two critical threats; the emergence of quantum computers and attacks against the software update supply chain. The outcome of this work will provide guidance to CI system designers and operators so that they can make an informed decision on the adoption of this combined approach to deliver a secure software update solution.

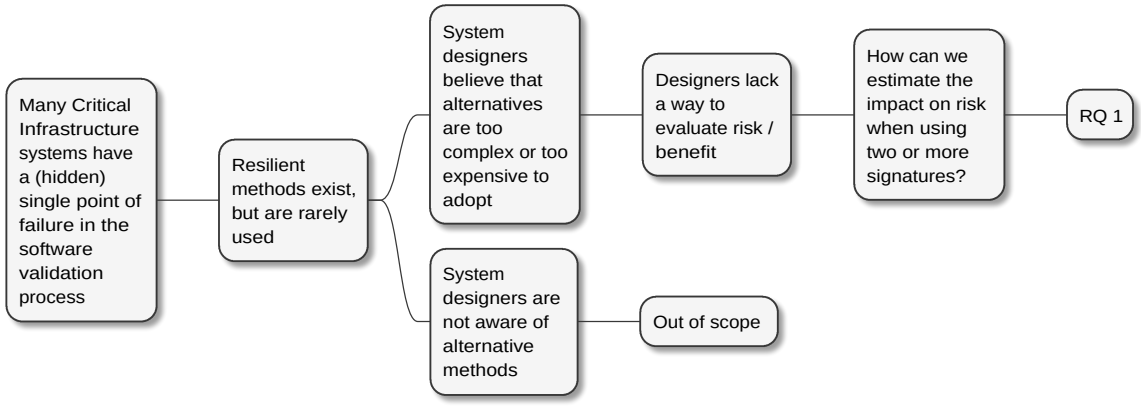

Figure 1.2: Mind map of problem statement.

1.3 Research Objectives and Hypotheses

Figure 1.2 is a mind map that explores the challenge of building systems that are resilient against supply chain attacks. Starting (on the left) with the statement that many systems have a single point of failure, it is observed that resilient methods (such as TUF, Software Updates for Internet of Things (SUIT), and Uptane) exist that are designed to mitigate this risk, but adoption of these methods is limited to just a few projects.

Two explanations are considered for the reluctance of system designers to adopt these methods. One is that system designers are simply unaware of alternatives. While there may be opportunities for better marketing and communication, a resolution to the issue of awareness is outside of the scope of this research. The other option is that designers have no means to compare the relative risk of using a single signature when compared to multi-signature options. This suggests that a means of measuring the relative risk may influence behavior. Therefore, the first research objective is to develop a model to show the relative impact on system risk when two or more signatures are used.

- *Research Question 1:* In what way does the probability of a successful attack change when multiple independent digital signatures are used to validate the integrity and authenticity of a software update package?

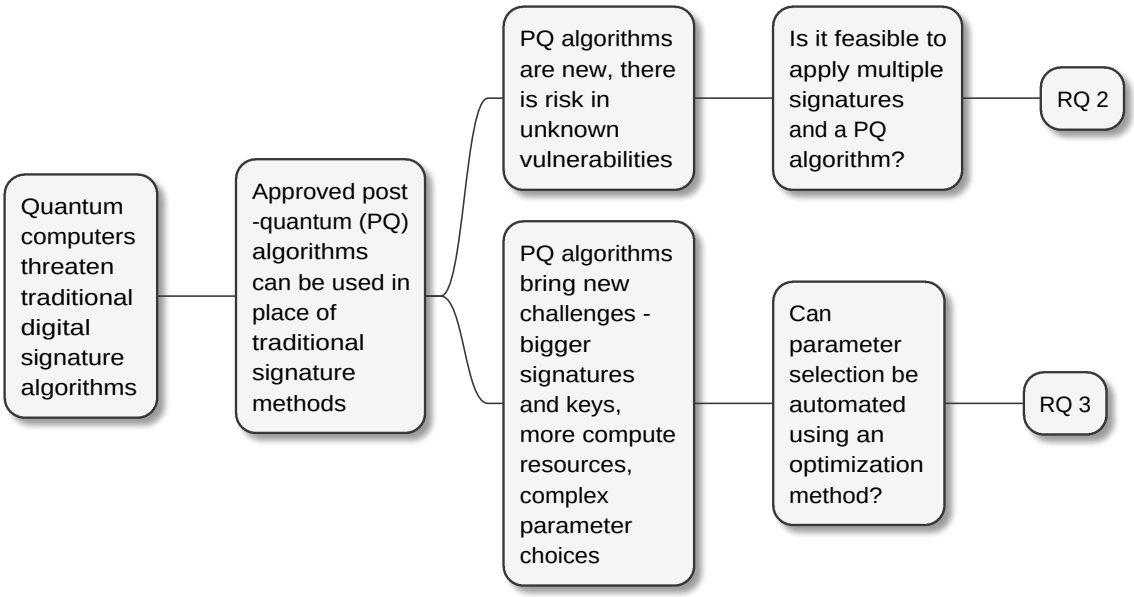

Figure 1.3: Mind map of problem statement (continued).

- *Hypothesis 1:* A statistical model will show that the use of two or more signatures significantly reduces relative risk of a successful attack when compared to a single signature.

Figure 1.3 explores the issue of post-quantum algorithm adoption. It is observed that there are existing, approved algorithms such as SHBS which can replace traditional digital signature methods today. This presents two challenges. First, these algorithms are new and may introduce new risks. It is therefore desirable to incorporate these new algorithms in the context of a high-resilience update framework. Due to the increased size and computational effort needed for post-quantum signatures, it is not clear if this is feasible. Thus, a second research question will focus on the system resource utilization when adopting an approved SHBS algorithm with the TUF update method.

- *Research Question 2:* How does the adoption of SHBS signatures change the network and system utilization of a client system when combined with the TUF update method?

- *Hypothesis 2:* Simulation results can be used to estimate the network and system

resource utilization required when using approved SHBS algorithms for many different parameter set choices.

The lower path in Figure 1.3 explores the challenge of selecting parameters for SHBS algorithms. There are many parameter sets that have been approved for use and system designers have little experience in deploying these algorithms. Therefore, it would be desirable to recommend an automated method for choosing an effective set of parameter options. The introduction of a method for selecting suitable parameters may further reduce the barriers to adoption of a multiple signature framework with post-quantum algorithms.

- *Research Question 3:* How can the complex problem of selecting parameters for a software update architecture be automated?
- *Hypothesis 3:* Genetic algorithm can be used to find an optimal (or near-optimal) set of parameters for a software update architecture with respect to network and system resource utilization allocated to the update process.

1.4 Research Contributions

This dissertation addresses the risk of software update for CI systems. It specifically explores the relative risk of adopting TUF which uses multiple signatures to validate the integrity of an update package. It also evaluates the network utilization and system performance impact when SHBS is used in combination with TUF. Finally, a method is recommended for supporting design decisions by automating the selection of complex parameter choices. These results are intended to assist system designers is making well-informed, risk adjusted choices when considering adopting the use of SHBS with TUF as a method for secure software updates.

1.5 Overview of Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** contains a literature review of software update methods and frameworks, selected attacks, and post-quantum algorithms.
- **Chapter 3** introduces the methodologies used to address each of the three research questions.
- **Chapter 4** applies a statistical model and shows the results of a risk model that can be applied to multiple signatures.
- **Chapter 5** examines the impact of incorporating an approved post-quantum algorithm in a multi-signature framework with simulation output to show the impact of complex system interactions.
- **Chapter 6** demonstrates results from the use of a genetic algorithm to support automated parameter selection with sample results.
- **Chapter 7** provides observations and conclusions with comments on future work.

Chapter 2: Literature Review

Figure 2.1 provides an outline of the literature that is relevant to the research questions that were defined in Chapter 1. An understanding of the software update process starts with a review of the current practices used in a secure software development, release, and support cycle. This is discussed in the context of the environment for software development and includes an introduction to the traditional methods for protecting software updates against attacks. That is followed by a review of the unique features required for software development in the domain of critical infrastructure systems.

In order to understand the challenges posed by software updates, it is helpful to also understand a historical view of actual attacks that have taken advantage of weakness in the software update mechanism. There is a review of several documented examples of exploits that compromised the software supply chain. This is followed by specific threats against critical infrastructure systems and recent trends.

Next, an introduction to post-quantum algorithms is provided. This section has a brief review of the emerging need to protect systems against future attacks that will apply quantum computing resources. This is followed by a review of algorithms that are believed to be resilient against quantum attacks and their attributes. This section introduces the specific

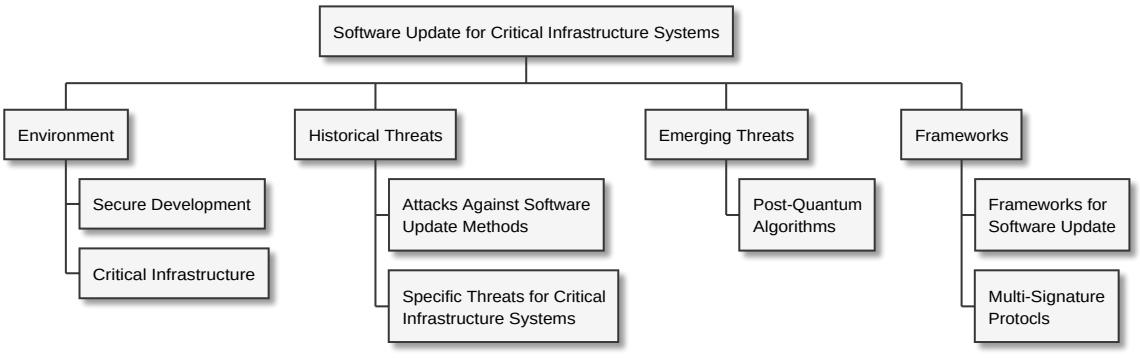

Figure 2.1: Outline of literature categories.

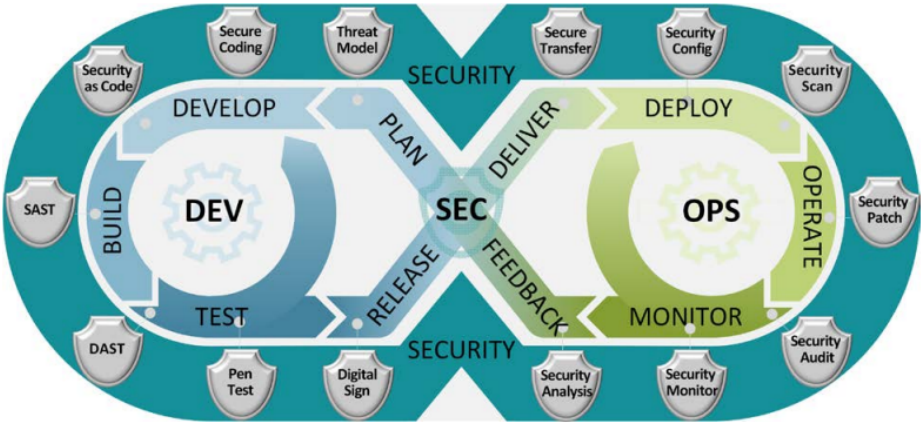

Figure 2.2: DOD DevSecOps software life-cycle (Lam and Chaillan, 2021).

recommendations from NSA for systems that will require long-term protection against quantum threats.

This is followed by an overview of software update frameworks. These are structures and methods that have been proposed to address known weaknesses in the software supply chain. This review includes an overview of the various types of multi-signature protocols that may be used to protect against various key compromise and collusion attacks.

2.1 Software Development Environment

The specific workflows and practices used by software development teams can vary greatly. One common feature is that there are generally two intersecting processes. One process is the design, development, and release of new functionality and new features. The other process is a learning and feedback loop that informs future development requirements. These two processes are often described as a lemniscate as shown in Figure 2.2 (Lam and Chaillan, 2021).

The *DoD Enterprise DevSecOps Reference Design* provides guidelines for software development teams to develop and maintain trustworthy software. Following the best practice, the left side of Figure 2.2 describes the process used to design and develop new features and new functionality. The right side represents the feedback loop from deployed

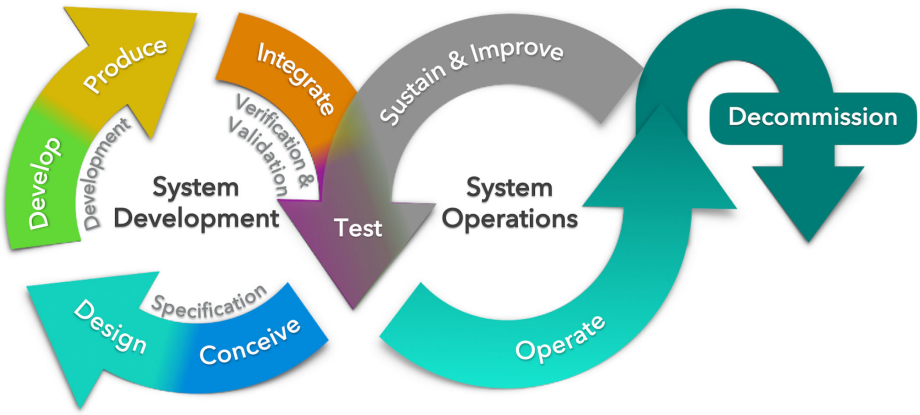

Figure 2.3: SECDevOps system life-cycle (Mathieson et al., 2021).

systems and the environment. This feedback process informs new requirements that drive future development and system improvements. Mathieson, Mazzuchi, and Sarkani (2021) adapted this approach into an Model-Based Systems Engineering (MBSE) framework by defining a new systems modeling language profile resulting in the modified full lifecycle diagram shown in Figure 2.3. The complete system lifecycle model is termed Model-Based System Operations (MBSO) and it adapts MBSE practices to the full system lifecycle.

In the DevSecOps and MBSO models, software updates are validated and applied as part of the "Operate" phase. This occurs after a system is deployed and in operation making it difficult to ensure that newly released software is authentic. It is the gap from the "Release" process to the "Operate" phase that represents an opportunity for threat actors to introduce modified and potentially malicious software into a system.

This gap is identified in National Institute of Standards and Technology (NIST) Special Publication 800-160, *Developing Cyber-Resilient Systems: A Systems Security Engineering Approach* (Ross et al., 2022). Specifically, this document states that "System modifications and upgrades can also introduce additional vulnerabilities." Protections against modification of an update package are typically applied as part of the "Release" and "Deliver" phase of the system lifecycle. NIST identifies these lifecycle phases as potential areas of attack, stating that "those systems are an attack surface in addition to the interfaces and procedures by which updates are made to the system of interest" (Ross et al., 2022).

An approach for protecting systems against modifications in software update packages is provided by Cooper, Regenscheid, and Souppaya (2018). This white paper describes best practices for applying a digital signature to a software update. This method allows the target system to detect if any of the update files were modified or replaced after the "Release" process was completed. The authors define the concept of a Code Signing System (CSS) which is more than just a digital signature process. This system incorporates a process to validate that the code being presented for signing is actually valid and authorized for release. It also includes an audit trail to enforce accountability for all actions taken using the CSS private key. The paper also recommends the use of a trusted timestamp operation that can ensure that an accurate and authorized release time is included with each signed software package. Client systems that must verify the software can use this trusted timestamp to compare a new update package against the currently installed code and block installation of any older software. This approach protects against a "roll-back" attack, where an attacker presents an old update package to a client system. The old package may have a valid signature which makes it appear to be a valid update. However, the attacker may be aware of specific vulnerabilities that are present in the older version of the system code. If the attacker is able to coerce a client system to install an older update package they can force the system to roll back to a state where the known vulnerability was present.

The process for creating a digitally signed update package is further defined in the international standard ISO/IEC/IEEE 12207, *Systems and software engineering — Software life cycle processes* (Alm, 2017). This document defines operational requirements to enforce the long-term lifecycle of deployed software. This includes a periodic review of system protections against newly published security threats and the development of patches to reduce risk.

The process of signing software updates has become accepted as a best practice. Starting in 2002, Microsoft introduced a series of security improvements aimed at reducing the vulnerability of software directly produced by the company and software executed on their

operating systems. The Security Development Lifecycle (SDL) is a framework for software development was one of the significant outcomes of this process. Viega (2011) reviews the lessons learned through this process. This work points out that even with significant investment and focus on security, it is not possible to eliminate bugs and vulnerabilities in new software. A more effective approach is to provide regular and secure software updates that remove or mitigate threats as they are identified. It is also valuable to introduce system-wide features that minimize the impact of common threats such as buffer overflows and unallocated memory access.

Geer (2008) expands on the notion that threats can not be eliminated by observing that complexity is the enemy of good software security. This work compares the challenge of writing secure code to the challenge of managing the global economy. In both cases, the growing complexity of the environment is intractable, making it impractical to expect perfect performance in either domain.

Mohammadi (2014) takes this concept further. Recognizing that there is a real cost in reviewing, testing, and deploying software updates, this work develops a systems engineering assessment to prioritize which updates are most critical and strikes a balance between cost and risk. Enterprise Information Technology (IT) managers can apply this process to dynamically choose which updates to install based on a cost and risk model that aligns with their organizational requirements.

An alternative approach to eliminating software vulnerabilities is to motivate teams of developers to focus on identifying weaknesses through large-scale competitions. Parker et al. (2020) propose structured competitions to help with early identification of flaws in software systems. They examine the use of games with prizes for top scores related to creating secure services, finding bugs in services, and quickly fixing and updating vulnerable systems. They found that software written in C and C++ was 11 times more likely to have a reported security flaw than code written in a statically typed language. While effective in reducing vulnerabilities, this approach does not eliminate risk and it does not reduce the need for

software updates to correct for flaws discovered in systems that are already installed.

Avery and Wallrabenstein (2018) also recommend a game-based approach. Rather than encouraging teams to compete to identify patches, their method is to inject "deceptive" patches into code as a way to slow down attackers. These patches have the appearance of a legitimate software fix, but do not have any real impact on the execution of the software. They find that in limited cases, this practice can increase the resilience of a system against attack.

Not all software is developed by a single team following a formal process like SDL. Most modern software depends on libraries and functions provided by external teams with different practices. Gu et al. (2023) explore the threat of packages that can threaten the integrity of a software system. They compare several different public software repositories and show the ways in which attackers can inject malicious code with limited risk of being discovered. They classify six distinct ways that attackers can threaten the integrity of a secure repository. In their research, they disclosed 12 security threats, 8 of which had not been previously identified.

Many vulnerabilities found in software are merely "accidental" while others result from poor design or ineffective implementation of the software development process. Boyes (2015) describes how software is built from many components that may be sourced from different teams. This is similar to a physical product made of multiple components that are designed and manufactured by independent suppliers. The quality and reliability of the resulting system depends on the integrity of all components. In (Hatta, 2022) this concept is described as the "Nebraska problem". This work shows that many complex systems are dependent on underlying code components, some of which may be maintained by a small team of developers. Through either carelessness or bad intent, that small community of developers could threaten the integrity of a larger, more critical and more complex project.

There is a balance between trust and risk. Work by Nygård and Katsikas (2022) points out that increased trust in the form of a more formal process integration and contractual

relationship with a collection of suppliers is generally considered to be a benefit that reduces risk. However, increased trust in a targeted set of vendors can also increase risk in that it increases dependency on an external resource. A zero-trust model is recommended for all external providers of software components. They point out that "Cyber risk can be treated by mitigating it; avoiding it; sharing it; accepting it; or by a combination of these options."

NIST Special Publication 800-30 (NIST, 2012) provides guidance for conducting risk assessments on complex systems. This guidance makes 14 references to various risks related to threats to the system supply chain. This could take the form of disruptions to the original system delivery or in components required to maintain the system. A detailed risk assessment must consider the integrity of all of these sources of external risk.

The inability to eliminate security risks from software systems has resulted in a practice where it is accepted that frequent updates are necessary. As demonstrated by Mohammadi (2014), it may not be cost effective to install every software update immediately. This means that some systems will operate for a period of time without applying the latest security patch. Work by Coppens, De Sutter, and De Bosschere (2013) points out that the publication of a software update itself can give attackers an opportunity to identify the specific software defect that was corrected in the latest patch. They propose a method of intentionally modifying or obfuscating code to mask or hide the specific changes that have been applied as part of a new software update.

2.2 Critical Infrastructure Environment

The United States Department of Homeland Security (DHS) has defined a set of 16 CI segments that require special security and resilience considerations. The full list is defined in the National Infrastructure Protection Plan (NIPP) published by DHS (United States Department of Homeland Security, 2013) and provided in Figure 2.4. Disruption in any of these segments could harm the health, safety, and economy of the United States. These segments also involve a deep level of integration between digital controls and physical

Figure 2.4: DHS list of critical infrastructure segments (United States Department of Homeland Security, 2013).

assets in large-scale systems. This means that an effective cybersecurity attack that can compromise these systems can transform from a digital attack into a kinetic result.

The international standard ISO/IEC 62443 was created to define requirements to protect CI systems. Cindrić, Jurčević, and Hadjina (2025) provide an overview of the collection of documents that make up the 62443 standards. These standards include a required workflow for software update management. Table 2.1 provides a summary of the key steps in the process. NIST Special Publication 800-40 and 800-53 provide similar guidance in (Souppaya and Scarfone, 2022) and (National Institute of Standards and Technology, 2020).

Design of robust CI systems can benefit from extensions to conventional systems engineering processes. Cyber Oriented Digital Engineering (CODE) was introduced by Mitola and Prys (2024) to provide a repeatable systems-engineering process for systems to resist Advanced Persistent Threat (APT) cyberattacks. These are attacks that are executed by well funded and coordinated attackers such as nation state level threat actors. This method applies a Zero Trust (ZT) approach to designing CI systems. The ZT approach requires that system designers must assume that any system on a network may become compromised and must always validate externally provided content. This concepts extends to software updates although Mitola and Prys (2024) do not make any specific recommendations for protecting

Table 2.1: Software update stages from ISO/IEC 62443.

Identification	System operators need to be aware of new software updates when they become available for devices on their network.
Prioritization	It may not be feasible or desirable to immediately install all updates. A process is needed to prioritize the order of installation.
Testing	New software shall be tested in a controlled environment before it is distributed to a number of production devices.
Deployment	A managed process shall track the deployment of new software and record which impacted devices have successfully installed the latest software.

updates. The Systems Engineering DevOps (SEDevOps) process introduced by Mathieson et al. (2021) can also be applied to CI systems.

Ten, Manimaran, and Liu (2010) introduce a methodology for modeling attacks against CI systems. This work focuses on Supervisory Control and Data Acquisition (SCADA) systems used in the electric power grid which serve as a proxy for many other types of CI components. This work uses attack trees to analyze potential threat scenarios and then defines a set of "Vulnerability Indices" that can be combined to compare the net risk across multiple attack scenarios. Abdo, Kaouk, Flaus, and Masse (2018) point out that safety and security are typically evaluated and mitigated as independent activities. They combine attack trees with bowtie analysis to provide a comprehensive view of both safety and security risks to a complex CI system.

Another common feature of many CI systems is that they have limited resources in terms of computational capacity and storage. Zandberg, Schleiser, Acosta, Tschofenig, and Baccelli (2019) look specifically at secure firmware updates for resource limited Internet of Things (IoT) devices. This work describes the types of resource constraints that are typically encountered when applying cybersecurity solutions in a CI system. For example, they describe highly constrained systems with less than 32 kB of RAM and 128 kB of flash

memory. A traditional layered cybersecurity approach may not be practical in a system with such limited computational and code storage resources.

Paté-Cornell, Kuypers, Smith, and Keller (2018) point out that historical data is of limited use when analyzing risk in CI systems. In this industry, successful attacks are rare and reporting of both failed and successful attacks is inconsistent across the industry. They recommend looking at indications of probing activity to estimate the level of sophistication and maturity of potential attacks that may lead to high-consequence events. This work introduces three countermeasure techniques where there is an estimated optimal level of investment needed to mitigate attacks against a specific asset.

Baggott and Santos (2020) also look at risk analysis in the electric power grid. They point out that the power grid, like other CI systems, is complex and highly interconnected. Their risk analysis approach also looks to bridge between cybersecurity and safety. In this work they include risks due to natural disasters along side cybersecurity and equipment failure. This work considers a coordinated cyberattack against multiple elements of the electric grid in addition to a targeted attack on one resource. The nature of software updates which often use the same mechanism across many different devices may introduce an avenue for this type of coordinated attack. When considering such a distributed and coordinated attack, the authors emphasize that the resulting threat surface is vast and challenging to effectively analyze.

2.3 Historical Attacks Related to Software Updates

In December of 2020, cybersecurity firm FireEye noticed some unusual traffic on their corporate network. An internal investigation discovered that the activity was related to the Orion product from the software company Solar Winds. They found that the malicious software had been included as part of a routine software update that was released by the company in March, 2020. The update appeared to be perfectly normal with a valid digital signature indicating that it was valid code that had been shipped by Solar Winds to thousands

of clients including many US government agencies.

The Solar Winds attack from 2020 is one of the largest and most consequential examples of a software supply chain attack. Reaction to this attack has caused many cybersecurity experts to reconsider the balance of risk and benefit for widely distributed software updates. Peisert et al. (2021) and also Massacci, Jaeger, and Peisert (2021) present a breakdown of the implications of this attack on the industry. The scale of this attack made it clear that a threat to the software supply chain in one widely adopted software product can have immediate consequences for thousands of customers.

In March of 2023, the Money Message ransomware group infiltrated the corporate network belonging to Micro-Star International (MSI), a computer hardware manufacturer. The ransomware successfully extracted and downloaded a large batch of data from the MSI network. After MSI refused to pay a ransom, the attackers released all of the stolen data on the dark web. By May 2023, security researchers discovered that the leaked data included a copy of the private signing keys used by MSI to sign firmware updates for their products. With access to these keys, attackers could produce malicious firmware with a valid digital signature. Since these keys are permanently "burned in" to the company's products, there is no way to replace or update the keys used to validate future software updates. To this day, MSI systems built before the discovery of the breach are vulnerable to this attack. Jackson (2023) discusses this attack and shows that it is part of a growing trend in highly effective software supply chain attacks.

Ladisa, Ponta, Sabetta, Martinez, and Barais (2023b) use attack tree analysis to explore the many ways in which a software package can become compromised prior to distribution. This work is focused on attacks where malicious code is inserted into the legitimate build process. This is the method that was used in the case of Solar Winds in 2020. The analysis does not account for cases where the private keys used to sign valid code might be released as was the case with MSI in 2023.

One area of particular concern is the prevalence of open source software. Wermke et al.

(2023) provide results from 25 in-depth interviews with contributors to popular open-source software. They found that quality management and security protections vary significantly across projects. They point out that several open source projects are incorporated into the build process for many commercial packages. Therefore, an end user might be exposed to these risks without realizing that their systems are running code produced in an open source project. Levy (2003) examines this in more detail through an analysis of 8 different incidents where open source projects were temporarily compromised. The work by Hatta (2022) on ways in which an attacker might infiltrate a small and relatively unknown development team is relevant to this trend.

Ladisa, Plate, Martinez, and Barais (2023a) introduce a Systematization of Knowledge (SoK) level taxonomy of attacks on open-source software supply chains . A similar SoK on software supply chains is provided by Nygård and Katsikas (2022). These works demonstrate the high level of sophistication that is possible with complex supply-chain attacks and the need for secure mechanisms for software distribution and validation.

Digital signatures are used to validate that the content of a software update is authentic and unmodified. Attacks on digital signatures and the keys used to sign software represent an entirely different way for software updates to be compromised. Hernandez-Ardieta, Gonzalez-Tablas, de Fuentes, and Ramos (2013) provide a survey of attacks on digital signatures, showing that threats to a digital signature workflow can erode confidence in the authenticity of the signature. Lax, Buccafurri, and Caminiti (2015) provide an alternate perspective on vulnerabilities in digital signatures related to the "unobservability" of the signing process. In the context of software updates, this relates to the process where a signing system accepts a software update file and produces a signature. The machine-centric nature of this process makes it difficult for a system operator to be confident that the signature has been applied to the intended binary code. If an attacker were to modify the signing process, it may not be noticed by manual or automated systems in the workflow. Yu (2016) explores options for protecting private keys from compromise as well as recovery in cases where an

attack is suspected.

2.4 Attacks Against Critical Infrastructure Systems

Early in 2010, operators at the Natanz uranium enrichment facility in Iran started to notice that many of their centrifuges began to spin much faster than intended and subsequently the equipment began to fail. It is estimated that up to 1,000 devices were impacted, causing a significant disruption to the nuclear development program. These mechanical failures were the result of a malicious software update to the Siemens Step 7 software that implements the closed loop control of the centrifuge spin rate. This update came to be known as "stuxnet". Nourian and Madnick (2018) provide a detailed analysis of this attack. It was the first well known and publicized case of a targeted attack on a CPS where digital malware caused a significant physical outcome.

Schneider Electric Triconex safety controllers are frequently used in CI systems such as oil and gas pumping and refinery systems. These systems have physical buttons that can be used to initiate a system shutdown if an operator detects unsafe conditions. The TRITON malware (sometimes called TRISIS) inserts new code in the interrupt handler that responds to these buttons allowing remote commands to initiate an unplanned shutdown. TRITON is the first known case of a safety system being weaponized through malware to become a threat to system integrity. Geiger, Bauer, Masuch, and Franke (2020) describe this attack along with earlier attacks called "BlackEnergy 3" and "Crashoverride".

In 2016, operators at Ukrainian power plants began to notice unauthorized activations of circuit breakers causing power disruptions. Their systems had been infected by the "Industroyer2" malware. Knapp (2024) provides a technical review of this attack and shows the relationship with the prior TRITON attack. This works includes an analysis of the trends in critical infrastructure malware which indicates that the software supply chain is a common method for injecting malicious code into CI systems.

Bencsath, Pek, Buttyan, and Felegyhazi (2012) point out that while Stuxnet received

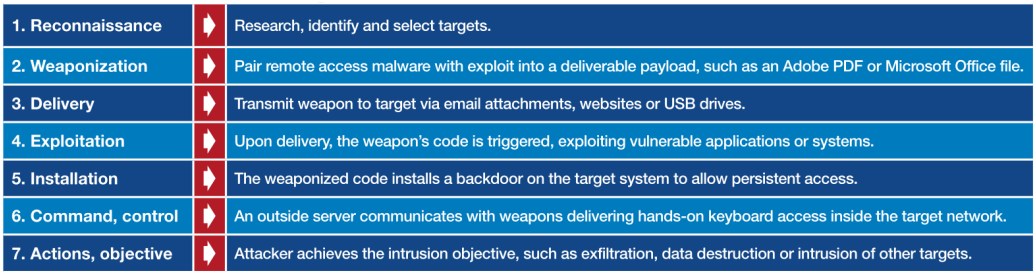

Figure 2.5: Seven phases of the intrusion kill chain for CI systems (Capano, 2019).

global attention, it was actually part of a series of attacks that began with earlier threats named "Duqu", "Flame", and "Gauss". While these attacks had far less impact, they show a trend toward progressively more sophisticated attacks that specifically target CI systems.

Bouramdane (2023) looks across the history of published attacks specifically on smart grid infrastructure. This background is used to develop a multi-criteria decision-making approach to prioritize cybersecurity defense options. Recommended areas for investment include network segmentation, patch management, threat intelligence, vendor and supply chain security. In the analysis, the digital supply chain is identified as both a threat and as a countermeasure. The author shows that some past attacks were distributed through vendor managed supply chain processes while security fixes and countermeasures can also be delivered through this same process.

Capano (2019) shows that attacks on CI systems are not just an event that occurs at a single point in time. Each attack has a prolonged lifecycle that typically passes through seven phases that can be referred to as an "intrusion kill chain". These phases, shown in Figure 2.5, are a sequence of repeatable steps. The time for each phase can vary significantly.

2.5 Post-Quantum Algorithms

Rapid developments in quantum computing technology are introducing new threats against traditional cryptographic mechanisms. The public key algorithms that support traditional digital signatures which protect software update packages against substitution or modifica-

tion may soon be susceptible to compromise by anyone with a sufficiently capable quantum computer (Buchmann and Ding, 2008). The application of Shor's algorithm provides a powerful technique that can be used to dramticially reduce the time needed to solve discrete log and elliptic curve related problems (Rieffel and Polak, 2011 - 2011). It is helpful to explore the literature on post-quantum options when proposing a path forward for CI software updates.

In 2022, the NSA published an update to the set of cryptographic algorithms that it recommends for security applications. The guidance, known as the CNSA Suite 2.0 recommends replacement of the Elliptic Curve Cryptography (ECC) and public key algorithms that were previously recommended to protect commercial and CI systems (National Security Agency, 2022). This guidance was updated in April 2024 with additional guidance in National Security Agency (2024).

The guidance introduced a choice of two SHBS algorithms. The two recommended algorithms are eXtended Merkle Signature Scheme (XMSS) and Leighton-Micali Hash-Based Signatures (LMS) which are further defined in NIST Publication 800-208 (Cooper et al., 2020). Implementation details for XMSS were initially introduced in the Internet Engineering Task Force (IETF) Request for Comment (RFC) 8391 (Huelsing et al., 2018) and LMS was introduced in RFC-8554 (Josefsson and Schaad, 2019).

While the CNSA Suite 2.0 is intended to phase-in over time, the document makes a specific and more urgent recommendation for software and firmware signing: "Begin transitioning immediately, support and prefer CNSA 2.0 by 2025, and exclusively use CNSA 2.0 by 2030." This guidance follows earlier advice from the NSA that caused some in the cryptographic community to question what the NSA really knows about the state of development of quantum computers and the real urgency of adopting post-quantum algorithms. Koblitz and Menezes (2016) suggest that the urgency of this guidance could indicate that NSA has more insight about the maturity of quantum computing technology than they can reveal publicly.

The use of SHBS requires careful management of system state. A detailed analysis of the unique requirements for state management is introduced by Chen, McGrew, and Mitchell (2016). This work discusses the inherent challenges of safely implementing state management in a software-only or any virtual environment where the system state may be reset to a previous condition. Challenges of key backup and recovery are also explored along with some hardware-based attacks such power line manipulation.

Butin (2017) presents a survey of the various digital signature algorithms that were believed to be quantum safe in 2017. This survey includes XMSS and LMS as well as SPHINCS, a stateless hash-based algorithm. Similar to the NIST recommendations, this survey recommends the use of hash-based algorithms for future applications due to their minimal security assumptions and generic applicability across platforms. This work also calls out the need for guidance for application developers to choose parameter sets for specific use cases, such as software update in embedded systems.

The two SHBS algorithms included in CNSA Suite 2.0 each offer an extensive list of approved parameter sets. Selection of parameter options is typically done using a heuristic approach. A linear optimization technique to derive parameter sets for the multi-tree variant of XMSS, known as XMSS-MT was introduced by Hülsing, Rausch, and Buchmann (2013). This technique is effective for generating individual parameter sets that meet one specific performance objective. However, the same approach can not be extended to picking among a set of approved parameters for use in a specific application or use case.

Gazdag, Friedl, and Loebenberger (2019) consider XMSS based signatures for software updates in a cloud-based repository. They recommend a hardware-based key server that manages the XMSS key state with a secure connection to a remote development system that must submit requests for signature. They also suggest a hybrid of DSA/ECDSA along with XMSS to support both current and future system requirements. This work examines 4 parameter options, all with a tree height of 20.

The transition to post-quantum cryptographic algorithms typically has a significant cost

in terms of system performance. A detailed comparison of LMS and XMSS implementations on an ARM Cortex-M4 processor is introduced by Campos, Kohlstadt, Reith, and Stöttinger (2020). This work considers 9 different parameter choices for performance testing, although several of the tested options are not approved in the NIST 800-208 guidance which came out after this paper was published. The authors show modifications to the standard algorithm that can achieve performance improvements on the target architecture. These modifications further deviate the results from the NIST approved algorithm set. A detailed performance assessment of quantum-resistant software update security on low-power networked embedded devices is introduced by Banegas, Zandberg, Baccelli, Herrmann, and Smith (2022). This work looks specifically at the IETF SUIT protocol and provides results for implementation on three different 32-bit microcontrollers. The digital signature algorithms selected for the simulation include LMS, Falcon, and Dilithium. In the case of LMS, only one parameter set is considered for performance testing. The specific implementation of LMS that is used in the test uses a 2-tree Hypertree Signature Scheme (HSS) variant.

Both stateful and stateless hash-based signatures are explored by Wagner, Oberhansl, and Schink (2024) for use in software update and secure boot in embedded systems. This work assumes a device lifetime of 40 years and a worst-case scenario where there are up to 2 software updates per-day during the entire life of the device. These assumptions narrow their choices of stateless algorithm parameters. This work includes a valuable analysis of the impact of the Winternitz parameter on the trade-off of signature size to performance.

A summary of the NSA algorithm classification and approval process is provided by Baseri, Chouhan, and Hafid (2024). Like many other surveys of post-quantum algorithms, this work is focused on network applications that require high-frequency digital signatures and encryption. It is silent on applications that require high-integrity signatures such as signatures used to protect software updates.

2.6 Software Update Frameworks

The most basic form of a software update or patch is a collection of new files that are intended to replace files on a system that is operating in the field. Distributing a set of individual files is inconvenient and highly error prone. Not to mention that some files might be modified, added, or removed before they reach a target system. A method of combining and labeling files that are part of an update package can be described as an update "framework".

An example of a simple form of an update framework is to collect files into a single ZIP or Tape ARchive (or TAR) file. These are file formats that allow for multiple individual files to be collected into a single file or "archive". The TAR format is defined in (IEEE and The Open Group, 2008) and the ZIP file is defined in (PKWARE, Inc., 2024). The diff output format, also defined in (IEEE and The Open Group, 2008), is another method that can define a consolidated set of changes needed to modify an existing set of files such that they match a target set of files. In the typical case where the number of changes required are small compared to the size of the original file set, this can be a highly efficient way to represent just the changes.

While these standard file formats can collect multiple files into one package, they do not provide a way to validate the authenticity of an update nor do they protect against the addition or modifications of files in transit. They also lack any way to check to see if the updated files are compatible with the target system. These more advanced features require a framework that describes the process for packaging, validation, and applying the modifications.

Okafor et al. (2024) provide a SoK review of several methods for packaging software updates with analysis on the supply chain security properties of each method. They define a software supply chain as "a collection of systems, devices, and people which produce a final software product". This work is focused on the enforcing validity and role separation in the

software build and testing process as well as tools that enable transparency in the artifacts that they produce. This includes a discussion on the benefits of including the generation of a Software Bill of Materials (SBOM) as a software build artifact. They develop an analysis framework that is focused on overall transparency, validity, and separation and use this to compare methods. This work provides detailed review of the following supply chain security models:

1. Supply Chain Integrity Model (SCIM) from Microsoft
2. Supply-chain Levels for Software Artifacts (SLSA) from Google
3. Cloud Native Computing Foundation's (CNCF) Software Supply Chain Best Practices
4. in-toto - a research project focused on reproducible builds
5. Sigstore - a set of practices that identify all sources used to create a build

Samuel, Mathewson, Cappos, and Dingledine (2010) point out that just applying a digital signature on an update package is not sufficient for long-term security. They show examples where the signing key used to protect update packages might be compromised. In a traditional update system, there is no way to remotely recover from this type of attack. They develop a new model that they call TUF. This framework separates the update process into several distinct roles and allows for signing keys to be replaced by a higher level role if they should be compromised. The option to have multiple signatures for each role allows for further protection against key compromise or the actions of a careless or malicious operator in the software supply chain. This update process is discussed in more detail in the TUF tutorial in Section 5.2.

The TUF model has been adapted for use in several specific applications. The IETF has defined SUIT which incorporates some of the features of TUF, such as the role of a file manifest to define a complete package of changes required for a complete update. Zandberg, Schleiser, Acosta, Tschofenig, and Baccelli (2019) examine the performance of SUIT on

several resource constrained hardware devices. However, this work does not consider the impact of post-quantum algorithms or the value of multiple signatures for each role. An alternate form of TUF for IoT devices is "UpKit", proposed by Langiu, Boano, Schuss, and Romer (2019). This variant represents a significant simplification of the overall TUF model which eliminates many of the features that lead to resilient systems. Another variation on TUF is called "Uptane". This is a specific variation of TUF that has been adapted for use in automotive software updates. Boureanu (2023) provides a detailed analysis of the structure of Uptane.

In spite of the strong benefits of TUF and the variants that have been defined for specific operating environments, adoption of this framework has been limited. Fischesr, Trummova, Gajland, Acar, Fahl, and Sasse (2024) discuss this trend where solutions that can enhance security are often slow to be deployed. They conducted 21 interviews with leading experts in cybersecurity fields about the pace of adoption. They find that adoption rates are hampered by miscommunication and misunderstanding among cryptographers and engineers and missing responsibilities and resources, as well as misaligned goals and incentives. Each of these represent complex organizational and industry-wide challenges.

2.7 Multi-Signature Protocols

In order to understand applications where multiple signatures can be applied, it is useful to consider standard signature structures where redundant, concatenated signatures can be used. The following sections describe three such protocols along with suggestions for applications where an application of the risk analysis methods introduced here may be applied. TUF is an example of one framework that supports multiple signatures. The following sections describe additional use cases where multiple signatures can be applied.

2.7.1 Cryptographic Message Syntax

The Cryptographic Message Syntax (CMS) is a modern adaptation of the Privacy Enhanced Mail (PEM) format that was initially created to support secure email. The current CMS structure is defined in RFC-5652. The underlying technical structure of a CMS signature object is defined using the ASN.1 syntax as defined in Housley (2009).

The ASN.1 code that associates a signature to a file or other digital object can be easily extended to support the concatenation of multiple signatures. However, the policy defined RFC-5652 states that a CMS object is only compliant if it contains a single signature. This requirement is necessary because the policy does not define how a recipient shall interpret multiple signatures.

2.7.2 Trust Anchor Management Protocol

Trust Anchor Management Protocol (TAMP) was developed to support the long-term management of roots of trust in a distributed system. In many cases, the "trust anchors" in the system include the public keys used to validate safety-critical messages including software updates. The TAMP protocol recommends that updates to these critical parameters should not be included in a routine software update. Systems that use the TAMP protocol have a separate authentication process with independent signers that should be reserved exclusively for updating trust anchors.

The TAMP protocol is defined in RFC-5934. Similar to CMS, the content of a TAMP signed message is defined as an ASN.1 structure that can be extended to support multiple, concatenated signatures. However, RFC-5934 specifies that a valid TAMP message must have only one signature present. There is an exception in that TAMP can support an additional "contingency" signature that is used only if the primary signer has become compromised or unavailable (Wallace et al., 2010).

The policy for TAMP can be extended to support multiple signatures and a pre-defined quorum. An example of such an extension can be found in the design of the Security

Credential Management System (SCMS) Brecht et al. (2016). This system was developed to support large-scale connected vehicle applications in North America where it provides multi-vendor interoperability in a safety-critical environment. It allows vehicles and roadside infrastructure equipment to rapidly validate the integrity and authenticity of safety messages that may indicate an impending collision or traffic signal violation. The SCMS distributes a list of Public Key Infrastructure (PKI) root certificates that must be trusted by all vehicles for this system to continue operations. Recognizing that the average age of a vehicle in North America is 10 years and that some cars remain in use for much longer, it was recognized that the SCMS will need to continue operations for many decades if it is to fulfill the intended mission of the system. The chosen design is based on ECDSA signatures and it includes a set of five "electors" which apply independently generated, concatenated digital signatures with a quorum of 3 valid signatures (2022, IEEE 1609 Committee). In addition, electors can sign a message that introduces new electors into the trust list, allowing for an ongoing sequence of certificate roll-over events. This concept will be explored in more detail in Section 5.2.1.

2.8 Multi-Signature Algorithms

The topic of multiple digital signatures has been the subject of substantial research resulting in a wide variety of signature types and standards. Terminology for different types of digital signature techniques is not necessarily consistent. Labels such as *threshold*, *aggregate*, *sequential*, *multi-party*, and *hybrid* are generally used to describe signature schemes with specific properties. In many cases, these methods seek to offer algorithms that are particularly efficient in terms of data transmitted or operations performed by the recipient as well as meeting specific requirements on privacy of individual signers.

2.8.1 Threshold Signatures

An example of a conventional *threshold signature* algorithm is presented by Tzeng, Hwang, and Yang (2004). This work presents a signature scheme based on the classic discrete logarithm problem. This method allows multiple signers to contribute to a signing process that generates a single signature for validation. Since the cryptgorgrpahic effectiveness of the discrete log problem is expected to be significantly weakened due to developments in quantum computers, this type of signature is not a strong candidate for any systems that must remain secure for a long time in the future.

2.8.2 Hybrid Signatures

Bindel and Hale (2023) discuss options for *hybrid signatures*. This is a model where a signer will apply multiple signatures using different algorithms. Rather than seeking redundancy in case of the signer's system being compromised, the goal of this approach is to survive an event where one of the underlying cryptographic systems is compromised. The authors stress that validation of a true hybrid signature must provide a mathematical link between the various algorithms used such that a recipient of a message is forced to check all of the individual signatures and evaluate the combined result. This precludes the development of a policy which may prioritize some signature methods over others. This approach ensures consistency in the approach to validation while a policy-driven approach may reach different conclusions on the same signature based on a policy that changes over time. In the case of TUF, the mix of algorithms and the method for combining results is left open to interpretation. A true hybrid approach may be used, or a mix of independent algorithms may be used with a well-defined policy which must be used to evaluate results. This latter approach would not meet the strict requirements of a *hybrid signature* as proposed by Bindel and Hale (2023).

2.8.3 Composite Signatures

Ounsworth, Gray, Pala, and Klaußner (2023) define a structure for *composite signatures*. This work is a draft IETF standard for post-quantum PKI. This structure attempts to formally define the validation policy that must be applied when evaluating a signature. This method avoids the strict mathematical integration required by Bindel and Hale (2023). This work could be integrated with a future version of TUF to create a more rigorous policy definition and enforcement for signature validation. Like hybrid signatures, this approach can combine both classical and one or more post-quantum signature into a composite result.

2.8.4 Sequential Signatures

Gritzalis, Preneel, and Theoharidou (2010) propose a signature scheme that is specifically designed to address software updates. They observe that updates are always released in a sequence. They argue that the software validation model should require that validation necessarily confirm that all software updates have been received and also processed in the correct order. They propose a signature scheme that proves not only the integrity of the current update, but also the validity and completeness of all prior software updates. This method could be integrated with a future version of TUF and the iterative nature of the approach is well suited for use with SHBS algorithms. However, Gritzalis et al. do not address the need for multiple independent signers to contribute to the validity of an update package.

2.9 Byzantine Fault Tolerance

The topic of Byzantine Fault Tolerance (BFT) describes a method of analysis for redundant and fault-tolerant systems. The notion of the Byzantine Generals Problem was first introduced by Lamport et al. (1982). They consider the problem of a group of generals who must reach consensus on whether to attack or retreat. Each general is separated from the

others such that they can only communicate through messengers, and some of the generals may not be loyal. They conclude that for any group of N generals, they will reach a reliable conclusion if less than $N/3$ of them are traitors.

This problem has been studied extensively resulting in many variations in how the problem is framed. For example, Probabilistic Byzantine Fault Tolerance replaces the notion of a general being strictly loyal or a traitor with a probability distribution (Avelas et al., 2024). Efficient Byzantine Fault Tolerance is another branch that looks for ways to significantly reduce the number of messages that need to be relayed among distributed nodes in large-scale fault-tolerant systems (Zhang et al., 2022). A summary of many variations on BFT can be found in Berger et al. (2023). This problem has found new interest because it has been related to the analysis of blockchain technology Wang and Nixon (2023).

The common theme among BFT models is that they focus on consensus where there are multiple participants that have some level of communication among the group. This is a good way to describe many redundant and fault tolerant systems, such as a blockchain or a computing platform with redundant nodes. This bears some similarity to the concept of multiple client systems that must decide if a new software update is valid as discussed in Chapter 4. The difference is that in the case of a software update package that is delivered to multiple nodes in a CI system, it is very rare that the client systems have any means of communicating with other nodes that must make the same decision. In the software update context, each client system must make an independent decision based on the evidence available in the form of one or more digital signatures. In some other context, such as a corporate network where there are many systems that must accept the same software update package at the same time and where they have a means of direct communication, it may be valuable to consider a BFT approach to this problem.

2.10 Chapter Summary

This chapter has reviewed the literature related to software update methods in conventional and CI systems along with a review of several attacks that have influenced the industry. It has introduced the environment in which CI systems operate which are often more restrictive in terms of computational resources and storage than a conventional enterprise system. These factors will inform the critical performance metrics that are considered in the system design introduced in Chapter 5 and the optimization technique proposed in Chapter 6.

Recent standards and guidance for the transition to post-quantum signature algorithms were also introduced. NSA has specifically recommended the use of SHBS for software updates on systems that may be in the field for a long period of time. Prior research on the unique properties of these algorithms has been introduced along with earlier work on integrating these algorithms with software update methods.

Several messaging protocols that can support multiple signatures were also reviewed. All of these mature protocols can benefit from a risk model that provides guidance to system designers when deciding how many digital signatures to use to protect critical information or assets. The model developed in Chapter 4 may serve this purpose. As outlined in Figure 2.1, this collection of topics covers the scope of prior work related to software updates for CI systems.

Chapter 3: Methodology

Each of the research questions that were identified in Chapter 1 can be addressed using a distinct methodology. Figure 3.1 shows the relationship between the research questions, the corresponding hypotheses, and the selected methodologies. This chapter will examine the research requirements for each question and describe the selected research methodology. The detailed implementation of each of the research methods will be discussed in the subsequent chapters.

3.1 Impact of Multiple Signatures

Examination of research question 1 requires comparison of the probability of a successful attack for cases where single digital signature is used against a case where two or more digital signatures are available to validate the integrity and authenticity of a software update package. This suggests a statistical method that is able to compare the probability of a successful attack under different conditions. It also suggests a need to define exactly what potential outcome states are possible when evaluating a software update package.

The corresponding hypothesis states that a statistical model will show a reduction in the relative risk of a successful attack when two or more signatures are used. Data provided by Grabowski et al. (2000) shows that estimating probabilities of rare events in large-scale, distributed systems is exceptionally difficult. This work shows that the small numbers of events, the inconsistent reporting of those events, and challenges in comparing the distinct conditions under which different events occurred make it impractical to estimate an actual probability of a successful attack. Paté-Cornell et al. (2018) reach a similar conclusion when looking at reported threats against CI systems in the electric power grid.

The first research question does not require calculation of an absolute probability for a successful attack. It is sufficient to show the relative change in probability for the case

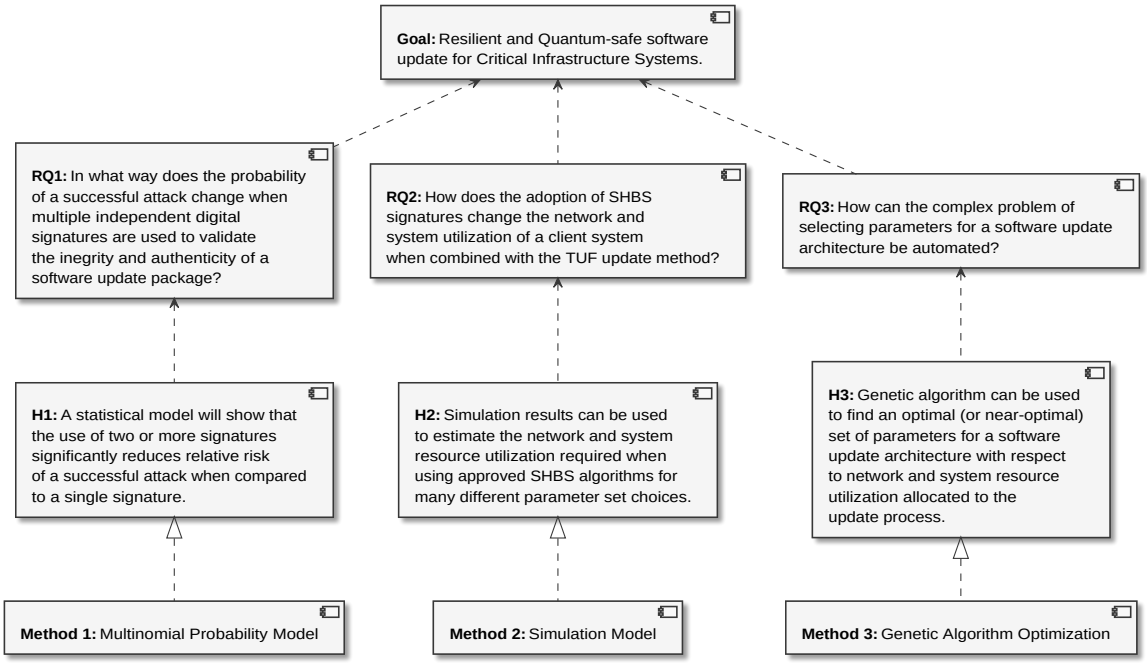

Figure 3.1: Linkage from research questions to methodologies.

where there is one signature or multiple signatures present. This can be done by choosing a representative probability for a rare event and comparing outcomes under several different conditions. This relative comparison requires enumeration of the potential outcomes of an attack and then build a probabilistic model to account for the aggregate probability of an undesirable result. A multinomial model is an effective means for this type of probability estimation.

The first research question will be evaluated by applying the following steps:

1. Define a set of discrete states that a client system can be in and the transitions between these states.

2. Classify the trustworthiness of a signing agent and define how this can impact client system state transitions.

3. Build a multinomial probability model that accounts for all possible outcomes from an update event.

4. Compare outcomes for several different system configurations and observe the relative impact of multiple signers on the probability of a successful attack.

3.2 Impact of Post-Quantum Algorithms on TUF

The second research question requires the examination of the impact of a specific set of approved SHBS algorithms when used with the TUF update method. The corresponding hypothesis suggests that simulation can be used to compare the impact for many different parameter choices. A simulation model built using the python programming language will be used to evaluate a sequence of updates representing the lifetime of two representative CI devices. The set of simulation results will then be analyzed to address the impact of the use of SHBS with TUF.

A detailed introduction to TUF is provided in Chapter 5. With this background it will become clear that predicting interactions among the multiple roles in this update framework is quite complex due to interactions among the roles in the update process. The problem is exacerbated by the fact that many CI devices are in the field for a decade or more, leading to a great deal of uncertainty in the sequence of updates that any one device will encounter. The logic used in building the simulation model will address this complexity and also demonstrate how the specific key management features in the TUF architecture can be utilized to overcome one of the perceived limitations in SHBS signature algorithms. The role of state management will also be examined through the structure of the simulation model.

3.3 Method for Automated Parameter Selection

The third research question requires the identification of a method for automatically selecting a preferred set of system parameters from a complex set of options. The structure of the simulation model will show that in the most general case, there are 4 distinct digital signature roles, each with 63 parameter options resulting in $4^{63} \approx 8.5 \times 10^{37}$ parameter combinations

that could potentially be considered for each system design. Rather than run the simulation over all of these options, it is desirable to identify an optimization method that can support system designers in choosing a practical set of parameters for a specific design.

It will be shown in Chapter 6 that there are two two performance criteria that may be important for a specific system design. These criteria are the net volume of data that must be communicated to all devices and the computational load required to perform signature validations, as required to support software updates over the system lifetime. Each of these criteria are impacted by the choices made for parameter selection in each of the software update roles.

The parameter selection problem is one where there is a discrete set of optional parameters for each of four roles. In this case the independent variable is not a smooth function that can be adjusted over a range, but instead is a set of discrete parameter choices where each option can have a significant impact on the desired performance metrics. The simulation model will show that there are also complex interactions among the four roles in the TUF method, so it is not practical to try to independently choose parameters for each role individually. The selection method must be able to evaluate options based on the impact to the complete system.

The hypothesis for research question 3 proposes that a genetic algorithm may be effective at selecting a preferred set of parameters. A genetic algorithm is suggested because this class of optimization techniques is well suited for discrete parameter selection problems with multiple optimization criteria. This method can evaluate many different combinations of discrete parameter choices where there is no significant structure to the parameter criteria.

Alternative discrete optimization methods such as branch and bound as well as particle swarm optimization were considered. These methods work better in situations where there is an assumed structure to the relationship between parameter values and the independent variables to be optimized. In the case of SHBS parameter selection, the sets of parameters are just a list of approved parameter values. A genetic algorithm has the potential to perform

better for this type of problem where it is reasonable to assume that there is limited structure among the parameter options.

3.4 Chapter Summary

Each of the three research questions posed in Chapter 1 will be evaluated using a distinct methodology. Research question 1, regarding the use of multiple signatures to validate a software update package, will be analyzed using a multinomial model to evaluate the relative probability of a successful attack when two or more signatures are used. Research question 2 which considers the combination of SHBS algorithms with the TUF update method will be explored using a simulation model. Finally, for research question 3 which calls for discrete parameter selection, a genetic algorithm will be used to determine if it is feasible to automate the process of recommending an approved SHBS algorithm parameter choice for each TUF role.

Chapter 4: Risk Model with Multiple Signatures

Like many complex systems design challenges, the development of systems that are protected against remote attacks by malicious actors introduces a complex balance of competing choices. As introduced in Table 1.1, some of the actions that may make a system more secure work against methods that make the system more reliable and available. In the case of CI systems where continuous operation is often essential to maintain safety, this introduces limitations on the security methods that can be effectively implemented. One technique that can help to mitigate this challenge is the use of multiple, redundant digital signatures to protect critical information such as software update packages.

Modern software update frameworks allow for the use of multiple digital signatures on update messages to protect systems in the event of the compromise or loss of a private key or compromise of the underlying digital signature platform. The choice of whether to use multiple signatures and the specific number of signatures to use should be based on a risk assessment. A multinomial probability model will be explored to evaluate the relative risk metrics for several different policy and system design options that include multiple signatures. The model will enable system architects and operators to make a risk-based decision when considering multiple signatures to protect critical software updates.

4.1 Software Distribution Workflow

Figure 4.1 represents a software distribution workflow with multiple independent digital signature operations. In this model the term *signer* is used to represent a system or agent that applies a digital signature to the released software update package. The model shows that there may be one or more signers who operate independently. A packager is shown which assembles the contributions from each signer into a consolidated update package. This file (or files) is sent over a distribution network to one or more clients. Note that each

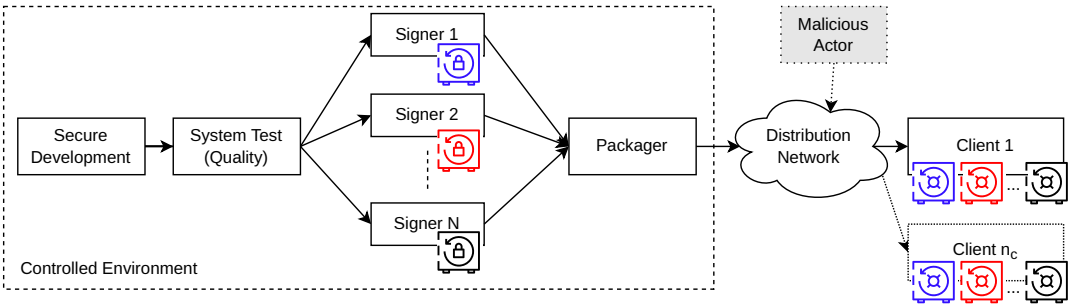

Figure 4.1: Simplified software distribution model.

signer has a single private key indicated by the vault icon.

The term *client* is used to describe a remote system that will receive a software update. Each client is provided with a list of all active and trusted public keys that correspond with the signer private keys. When a new update package is received by a client, it must decide if the update package can be trusted using the best information available. While multiple secure clients may receive the same update package, they typically lack any secure or reliable way to communicate with other systems, so each client must make this decision independently (Zhu and Leibowicz, 2022).

The movement from the packager to the clients may be a push or a pull operation meaning that an agent might deliver new updates directly to all registered clients (push) or each client might periodically check for updates at a known location (pull). No assumptions are made about the security of the distribution network. An attacker may be able to insert a false update into the channel as well as modify or block a valid update package before it is delivered to one or more clients.

4.2 Client Configuration

Client systems need to evaluate the integrity and authenticity of software updates that they receive. Specifically, software on the client must be able to validate that a new update package was created by an authorized system and that the package has not been modified. Trust is typically established by loading secure credentials on the client system during

manufacturing. Each of these credentials is typically referred to as a *trust anchor*. Each trust anchor includes at least one public key from a trusted software signing system.

Embedded devices may have limited secure storage. This may restrict the number of trust anchors that can be saved. Also, some digital signature algorithms require much larger key sizes than others. In particular, the most recent batch of post-quantum signature algorithms recommended by NIST all require significantly larger public keys than more conventional algorithms (Moody and Robinson, 2022).

The client trust store is typically initialized during manufacturing with key material that is under the control of the device vendor. A best practice is to ensure that the trust store can be updated over time using a secure update method that is independent of the normal software update path. For example, the TAMP protocol has been developed with special properties specifically intended for updating the list of trust anchors in a population of remote clients. Another best practice is to have a method of revoking an individual trust anchor. This is typically implemented using a Certificate Revocation List (CRL), although other methods may be used. For example, TAMP has a special message type that can be used to revoke a trust anchor. In Chapter 5, it will be shown that the TUF Root role can perform both of these functions.

4.2.1 Client Security States

From the perspective of software update status, a client system operates in one of three states:

1. Up-to-date
2. Vulnerable
3. Compromised

A client is in the *up-to-date* state if it has received and successfully installed all published security related software updates. This state does not imply that the system is protected

against all attacks, it just means that all available updates have been applied. It is assumed that software that has been released from a DevSecOps process will be protected against known or published threats.

A client is in the *vulnerable* state if there are security relevant updates that have not been installed. It is assumed that a security relevant update is intended to correct or patch a known or potential means of attack. Once an update is published, it is possible for attackers to use analysis tools to identify what has been changed in the software and this information can be used to identify a potential vulnerability in the original code.

A system can be in a vulnerable state for several reasons. Often this is temporary and caused by the delay in distributing, testing, and installing the new software on a collection of devices. For critical infrastructure systems, these delays can be fairly long due to the need to extensively test and validate that new software will not introduce flaws or errors in plant operations. Attackers can take advantage of this delay and attempt to exploit any vulnerabilities present in the updated software before the updates are applied to the vulnerable system (Coppens et al., 2013). The system may also be in a vulnerable state if an attacker chooses to block or interfere with the update delivery or validation process. An attacker may have a strategy to hold a target system in this state for a prolonged period of time, or they may just want to extend the time available to develop and deliver an attack.

A system becomes *compromised* when an attacker successfully executes an attack that introduces new code or injects some flaw or configuration change. There are many ways in which a compromise can occur. The following analysis will focus on a system that is compromised through the delivery and installation of a malicious software update.

4.2.2 Client State Transitions

A newly installed system running software that has no uninstalled updates will be classified as operating in the *up-to-date* state. When a new software update is published, the system immediately transitions to a state of being *vulnerable*, shown as transition 1 in Figure 4.2.

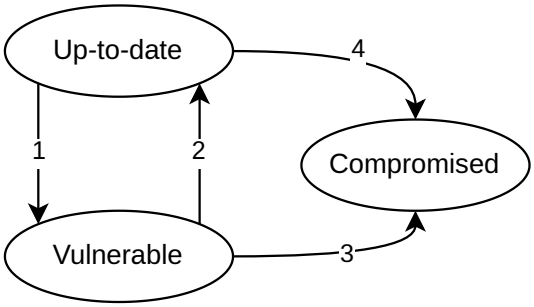

Figure 4.2: State transitions for a client system.

This is due to the fact that in releasing the software update, the publisher is effectively announcing to all attackers that there are specific vulnerabilities in the software that could potentially be exploited. Note that the due to defense in depth and practical constraints on the attacker's action, a system in this state may be effectively protected against the newly published vulnerability. However, the fact remains that it is now running software that has a known flaw. If an attacker were to assemble an appropriate "cyber kill chain", they could potentially exploit the vulnerability (Capano, 2019).

A *vulnerable* system can transition back to the *secure* state through the successful installation of an unmodified software update, shown as transition 2. Once this is done, the system should once again be protected against all known or published threats. A *vulnerable* system can also transition to the *compromised* state as in transition 3. This could be the result of an attacker successfully exercising an exploit, or it could result from the installation of a software update that has been compromised.

A system in the *up-to-date* state can also transition directly to *compromised* via transition 4. This would happen in a case where an attacker is able to insert a malicious software update into the distribution channel at a time when there is no official update available. From the perspective of any one client, it doesn't matter if the malicious software is sent through the normal distribution channel or if it is inserted directly at the point of delivery. If the client accepts the malicious software, it will immediately be compromised.

4.3 The Role of Signers

The signer role might be fulfilled by a fully automated system or it might involve one or more operators with special knowledge such as unique passwords or access to a locked room and cryptographic hardware. A signer may be part of a connected, online system or it could be a carefully protected offline component that is stored in a locked vault and only accessed when needed. For the purposes of this analysis, the specific nature and capabilities of the signers are not important.

In the simplified model shown in Figure 4.1, each signer is represented as an independent operation. This is a best practice, to operate each signer as an independent platform. For the purpose of efficiency, it is often tempting to combine resources such as secure storage or a secure execution environment for multiple signers. Any combination of operations among signers will reduce their independence and increase the potential that one attack may compromise multiple signers. Diversity among multiple signers can be achieved by using different hardware platforms, different operating systems, and different operational staff for each independent signing system.

4.3.1 Signer Workflow

The specific mechanisms or algorithms used by a signer will be generalized to a repeatable workflow. Every signer is assumed to execute the following steps:

1. Receive new content
2. Examine the content and make a judgment about validity
3. Calculate a digital signature that endorses the validity of the content
4. Deliver the digital signature to a client (directly or indirectly)

For many digital signature applications, the first two steps are inherently combined. New

content may be implicitly trusted because it was delivered through a trusted channel or as part of a workflow that operated within a secure environment.

4.3.2 Signer Threats

There are several ways in which a digital signing function can be compromised (Lesk, 2014). These can be captured in three general categories:

1. Cryptographic Threats
2. Platform Threats
3. Content Manipulation Threats

The threat that typically gets the most attention is an attack that exploits a vulnerability in the cryptographic operation used to calculate the signature. For many years, well established signature algorithms such as ECDSA and RSA have been considered to be secure in the sense that the best known attacks were impractical, even for a well-funded nation-state level attacker. However, with recent advances in quantum computers, this assumption must be revised. In the near future, it may become feasible to attack these formerly trusted algorithms, so it is reasonable to consider a situation where one or more signer's private key has been compromised.

A signing operation can also be threatened by an attacker who has access to the system or platform on which it operates. This could include physical or remote means of attack. An attacker who can access a signer platform can potentially command the signer to apply a signature to any content of their choice. Even if the attacker is unable to directly extract the private key, the ability to initiate the signing function and potentially inject content of their choice gives the attacker the ability to sign any file of their choice. A detailed taxonomy of digital signature attack types is provided by Hernandez-Ardieta et al. (2013). The more general categories presented here capture the full set of attack types that are relevant to the software update application.

In some cases, the attacker does not even need to access the system themselves. As with any employee, trusted operators of CI systems are susceptible to external influence or coercion. If an attacker can convince a system operator to sign a message that authorizes a malicious update, then they will achieve their goal without ever gaining direct access to the signing system.

An attacker may also focus on the transfer of software from the secure development and release process to one or more signer. If an attacker can insert malicious content at this point, or modify the legitimate content while it is being transferred, they can cause the impacted signers to apply a valid signature on malicious content, even if they have no direct access to the signer platform or the singer's key (Ladisa et al., 2023b).

4.3.3 Signer States

The previous section described several ways in which a code signing process may be compromised by a determined attacker representing an instance of an APT. For the purpose of modeling the risk of system compromise, it is valuable to consider the possible operational states of a code signing system. From the perspective of the software validation process a single signer can be in one of three states:

1. Trustworthy
2. Malicious
3. Unavailable

A *trustworthy* signer is one that will apply a valid signature only to a properly released software update. In this state, the code signing process will ensure that it only signs updates that are released through the approved SDL workflow. Any system in this state, by definition, has a private key that is not revoked or expired and has not been compromised.

A *malicious* signer describes a situation where the signing function has been taken over by an attacker. This can result from any of the attack types described in Section 4.3.2.

From the perspective of a client device that has received a signed software update, the specific mechanism of the compromise does not matter. A code signing system in this state can no longer be trusted to reliably sign only authorized update packages. It is further assumed that the system operators are unaware that one or more signing processes have been compromised. As soon as operators become aware of malicious activity, the correct reaction is to revoke the signing key as described below. It is also important to realize that in some cases a signing operation that is in the *malicious* state may continue to operate normally until the attacker is ready to execute a targeted attack. This behavior is examined in more detail in Section 4.7.

A signer is *unavailable* if it is unable to apply a valid signature. This could be the result of a system or workflow failure that resulted in the private key becoming inaccessible. While most secure signing systems do allow for backup and recovery of secret keys, a procedural failure or multiple system failures can result in a key being permanently lost. A signing key that is managed by a PKI system may also become unavailable if the signer's key has been added to a CRL. A signing system that has processed the CRL will recognize that they are listed and refuse to sign future requests. Any client that has accepted the CRL will also refuse to trust signatures from the revoked signer. This revocation process is the recommended best practice when system operators become aware that a signing system has been compromised. Another way for a system to become *unavailable* is for a PKI certificate associated with the public key to be expired.

The specific reason for the signer to be unavailable is not relevant to the client. All that matters is that the signer is no longer able to contribute to the validation of an update package. Note that the client will typically not be aware of the status of the signer. For example, if one of the trusted signers is revoked, the recipient of a signed message may not have access to the CRL that tells them this, or a signed update package might be delivered ahead of the CRL. All that the recipient knows is that one of the trusted signers that it knows about did not apply a signature to the current update. This could be routine or it could be an

exception. A client will not know if a signer is compromised if it has not been revoked.

4.4 Concatenated Signatures

The literature contains a wide variety of multi-signature and threshold signature mechanisms as introduced in Section 2.7. Many of these methods have the property that multiple parties can participate in generating a signature but the recipient only has to validate a single signature. These approaches are efficient in that the message that needs to be sent to the client is small and the computational workload for validation is minimal. However, many of these techniques impose requirements on each signer in terms of the types of signature algorithm that they can use and the calculations needed to combine the resulting signatures as shown in Tzeng et al. (2004) and Ding and Petzoldt (2017).

It is instructive to consider a very simple method for combining multiple signatures which can be classified as a *concatenated* digital signature. This method simply appends multiple, independent digital signatures in a single file or bundle. To better define this approach, consider that a message M must be delivered to a client. For example, this message may be a binary package containing a software update or a new set of operating parameters that must be communicated to a client with high integrity. It is assumed that M may contain parameters specific to the update operation, such as the time when the update shall be applied, a version identifier, or other information to be interpreted by the update mechanism.

A generic digital signature can be defined as in Equation 4.1. Here $S()$ is a general representation for a digital signature operation, M is the original message content, and K_i is a set of parameters specific to the signature operation. These parameters include a private key along with all details needed to fully define the signing algorithm. In this representation, there are no constraints on the algorithm selection. For example, K_i might specify a traditional public key algorithm such as ECDSA or it may call for a hash-based signature or a modern quantum-safe algorithm. The resulting signature, S_i is assumed to

be a binary representation of the result of the specific signature operation defined in the parameter set K_i.

$$S_i = S(M, K_i) \tag{4.1}$$

In the case where there are N signers, each signer will be associated with one parameter set. A collection of N signers would include the parameters $\{K_1, K_2, \ldots, K_N\}$. Each signer would independently generate a digital signature resulting in the set of values $\{S_1, S_2, \ldots, S_N\}$. A file containing this list of signature values is what will be defined as a concatenated signature. The complete package that will be delivered to the client would also include the original message M. The composite package M_c is defined as a set that includes the original message M and a collection of N signatures as shown in Equation 4.2.

$$M_c = \{M, S_1, S_2, \ldots, S_N\} \tag{4.2}$$

Note that the multiple signature approach described here is one where a conventional signature is simply repeated N times. Each signature is independently calculated and must be validated independently. This is different from conventional threshold schemes where the signature algorithm combines contributions from multiple signers to create a single signature (Shao and Cao, 2006).

Each individual signature S_i can be validated, producing a validation result R_i as shown in Equation 4.3. The validation function $V()$ takes the original message M and an individual signature S_i to produce a binary result. The value of each R_i can be `TRUE` if the signature S_i confirms the validity of M, otherwise it is `FALSE`. The overall evaluation of the validity of the message M from the collection of results in $\{R_1, R_2, \ldots, R_N\}$ requires a clearly defined policy.

$$R_i = V(M, S_i) \tag{4.3}$$

4.5 Validation Policy

When multiple signatures are used to valuate a message, a clear validation policy is needed to allow the client to decide if it should accept a new message. It will be assumed that the client is capable of understanding and validating all types of signatures defined in the set $\{K_1, K_2, \ldots, K_N\}$. This requires that the recipient has the capability of implementing the validation calculations in a secure environment and it has access to a trusted copy of any public information needed to support signature validation. It is further assumed that the client will evaluate each of the S_i signatures individually. Thus, for each of the signatures present in the message, the client will have a signature result R_i indicating if S_i supports the validity of message M or not.

Individual signers can sometimes be unavailable, as defined in Section 4.3.3. When this happens, the actual message delivered to a client may contain only N_c independent signatures, where $N_c \leq N$. It is also possible that some of the N signers will be revoked. When a signer is revoked, a client must ignore any signatures produced by that signer as if they were not present.

In order to accommodate unavailable and revoked signers, the validation policy for concatenated signatures includes a threshold or quorum value Q which is the minimum number of signatures that must be successfully validated in order for the client to accept the message. If the composite message contains at least Q valid digital signatures that all support the integrity of the original message M, then the client shall accept M and process the update or policy change that it contains.

It is important to note that in a case where the number of signatures present in a composite message is N_c and the policy defined quorum is Q with $N_c \geq Q$, it is not sufficient for the client to validate the first Q signatures and stop. A better policy is to require that the client always check all N_c signatures present. If any of the N_c signatures present in the composite message fails validation, then the client should reject the message, even if the number of

valid signatures equals Q.

The logic behind this policy is that there is no reason for there to be a mix of valid and invalid signatures in a trustworthy message. Consider a case where the total number of signers available is $N = 5$ and the quorum is set as $Q = 2$. In this case, at least 2 valid signatures are needed in order for a message to be accepted. It is possible that the message received may contain a full set of 5 signatures and only 2 can be validated and the remaining 3 fail. This result may indicate that an attacker has successfully compromised 2 of the signers and they have been used to sign some malicious content M_a while the remaining 3 trustworthy signers actually signed some other message M_v. If the content that was delivered contains the malicious update M_a instead of the valid content M_v, then accepting the 2 signatures that support M_a would result in the client being compromised. For this reason, it is required that client systems must always check all available signatures and reject any message with any signature that fails validation.

4.6 Software Vulnerabilities

When new software is released from a DevSecOps process, significant emphasis is placed on ensuring that there are no known vulnerabilities in the final product. Of course, this does not mean that vulnerabilities do not exist, but it is assumed that all *known* vulnerabilities have been address or mitigated in the final product.

As attackers and security analysts gain experience with a new software release, new vulnerabilities are frequently identified. Some of these new vulnerabilities may become publicly known while others will remain *unpublished* until they are used. The emergence of a previously unknown threat is known as a "zero-day exploit" and patient attackers will often carefully protect this knowledge and attempt to leverage it when it can be most effective (Paté-Cornell et al., 2018).

When software is installed and used, the operator takes responsibility for a combination of options selected and parameters that are set in the software. The local configuration may

introduce vulnerabilities into the system that are independent of the software itself. It is also typically the case that most systems rely on a collection of different software packages developed and maintained by different providers and released at different times. This means that as a system, a single device or function may have a mix of software at different states of maturity with a collection of known and some unpublished vulnerabilities (Capano, 2019). For the purpose of understanding the impact of multiple signers on the vulnerability of a system it is sufficient to consider only the vulnerability of individual applications.

4.7 Attacker Model

In developing a software update mechanism for a critical infrastructure application, it is important to consider that attacks may come from advanced persistent threats such as nation state actors. This type of attacker is generally both patient and strategic. They can afford to invest significant effort to execute complex, multi-stage attacks. It should also be assumed that they will be careful not to revel their presence in a system until they have high confidence that an attack will be successful (Levy, 2003).

When multiple signers are used and the system policy requires a quorum of at least Q signatures to accept a new update, then it is in the attacker's interest to ensure that they can compromise at least Q signers before they can execute a successful attack. In the case where they have access to less than Q signers under their control, it is likely that a strategic attacker will not want to reveal their presence in the system, so they will allow the compromised signers to continue to produce valid signatures on authentic updates. Only after the attacker is able to directly influence Q or more signers will they be able to execute a successful attack.

4.8 Risk Model

The use of multiple signatures introduces a risk trade-off. A signer can exist in three states, as introduced in Section 4.3.3: *Trustworthy*, *Malicious*, and *Unavailable*. In a case where

the number of signers is greater than the quorum required by policy, the additional signers can mitigate against signers being unavailable by providing redundancy. In the case where the minimum quorum is greater than 1, the risk of an individual signer being compromised is mitigated. It would be helpful if the choice of a specific value for N, the number of signers in the system and Q, the minimum required number of signers on a valid message could be informed by the relative risk of compromise or unavailability of a signer.

Consider a probabilistic model where a collection of N signers are operated independently. If all of the independent signers have equivalent security protections, then the value p_T can be assigned to represent the probability that any one signer is in a trustworthy state. The value p_M will represent the probability that a signer is malicious and p_X to represent the probability that a signer is unavailable with the constraint that $p_T = 1 - p_M - p_X$.

In assigning approximate values for p_M and p_X, the model can be used to estimate the aggregate probability of an update message being *valid*, *malicious*, or *unavailable*. To build the model, consider that N signers can be drawn from a population with replacement and create a table of all of the ways that N signers can be constructed using trustworthy, malicious, or unavailable elements. The probability of any one combination is calculated using the multinomial distribution which has the probability mass function shown in Equation 4.4.

$$P(n_T, n_M, n_X | p_M, p_X) = \frac{(N)!}{n_T! n_M! n_X!} (1 - p_M - p_X)^{n_T} {p_M}^{n_M} {p_X}^{n_X} \tag{4.4}$$

Here, n_T is the number of trustworthy signers, n_M is the count of malicious signers, and n_X is the number signers that are unavailable, such that $n_T + n_M + n_X = N$.

Consider an example where $N = 3$. Table 4.1 shows the list of combinations of signers such that $n_T + n_M + n_X = 3$. The right side of the table shows the likely result if a recipient were to process a message produced by each set of signers under policies where $Q = 1$, $Q = 2$, or $Q = 3$. The letters on the right side can be interpreted as follows:

- V: *Valid*, a valid update will be accepted by the receiver

Table 4.1: Example for $N = 3$ signers.

Row	n_T	n_M	n_X	$Q = 1$	$Q = 2$	$Q = 3$
1	3	0	0	V	V	V
2	0	3	0	C	C	C
3	0	0	3	D	D	D
4	2	1	0	C	V	V
5	2	0	1	V	V	D
6	1	2	0	C	C	V
7	0	2	1	C	C	D
8	1	0	2	V	D	D
9	0	1	2	C	D	D
10	1	1	1	C	V	D

- C: *Compromised*, an attacker may deliver a modified or malicious update that will be accepted by the recipient
- D: *Deferred*, the update will be deferred or skipped because there are not enough trusted signers available to validate the package

To better understand the attack scenarios and outcomes, it is helpful to examine several rows of Table 4.1 in more detail.

- Row 1: Here $n_T = 3$ which means that all 3 of the signers are trustworthy. In this case, for all values of Q, the recipient will be able to accept a *Valid* update.
- Row 2: Here $n_M = 3$ which means that all 3 signers are under the control of an attacker and may be malicious. For all values of Q, the attacker can successfully deliver malicious code and it will be accepted by the recipient. As a result, the client will be *Compromised*.
- Row 3: With $n_X = 3$, this represents a case where all 3 signers are unavailable. For all values of Q, the update package will be rejected or *Deferred*. In this case the recipient may be vulnerable to known attacks that would have otherwise been averted had the update been applied.

- Row 4: Here $n_T = 2$ and $n_M = 1$, so an attacker only controls one signer. With a policy that sets $Q = 1$, the attacker may successfully deliver malicious code and have it accepted, so the client may be *Compromised*. Otherwise ($Q = 2$ or $Q = 3$) a *Valid* update will be accepted. Note that for $Q = 3$ a strategic attacker will likely choose to instruct the signer that they control to produce a trustworthy signature so that they do not call attention to the fact that they control one of the signers. This provides more time to attack the other two signers.

- Row 5: With $n_T = 2$ and $n_M = 0$ there is 1 signer that is unavailable. With $Q = 1$ and $Q = 2$ a *Valid* update will be accepted. With $Q = 3$ updates will be *Deferred*.

- Row 6: With $n_T = 1$ and $n_M = 2$ the attacker now controls 2 signers. With $Q = 1$ and $Q = 2$ the attacker can choose to deliver malicious code and the client may be *Compromised*. With $Q = 3$ the attacker will instruct their signer to agree with the trustworthy signers and a *Valid* update will be accepted.

- Row 7: Here $n_T = 0$ and $n_M = 2$ so the attacker controls 2 signers and the third is unavailable. With $Q = 1$ and $Q = 2$ the attacker can choose to deliver malicious code and the client may be *Compromised*. With $Q = 3$ the update will be *Deferred* because there are only 2 signers available.

- Row 8: With $n_T = 1$ and $n_M = 0$ there is only one valid signer available. With $Q = 1$, a *Valid* update will be accepted. For $Q = 2$ and $Q = 3$, the update will be *Deferred*.

- Row 9: With $n_T = 0$ and $n_M = 1$ the attacker controls just 1 signer and 2 are unavailable. For $Q = 1$, a malicious update may be accepted and the client may be *Compromised*. For $Q = 2$ and $Q = 3$, the update will be *Deferred*.

- Row 10: With $n_T = 1$ and $n_M = 1$ there is 1 trustworthy signer, 1 that is malicious, and 1 that is unavailable. For $Q = 1$, the attacker may deliver a malicious update and

have it accepted, so the client may be *Compromised*. With $Q = 2$, the attacker will wait and only *Valid* updates are accepted. For $Q = 3$, updates will be *Deferred*.

The net probability of a specific outcome can be calculated by adding the probabilities of the signer combinations that contribute to each outcome. For example, in the case of $Q = 2$ there are three combinations that result in the recipient becoming compromised on rows 2, 6, and 7. The net probability of the recipient being compromised for $N = 3$, $Q = 2$ as shown in Equation 4.5.

$$\begin{aligned} P_C(N = 3, Q = 2) = P(n_T = 0, n_M = 3, n_X = 0) + \\ P(n_T = 1, n_M = 2, n_X = 0) + \\ P(n_T = 0, n_M = 2, n_X = 1) \end{aligned} \tag{4.5}$$

Sums of this sort can be repeated to calculate the net probability for each outcome resulting in a net probability of the recipient accepting a valid update (P_V), the probability of being compromised (P_C), and the probability of the update being deferred (P_D) for any combination of N and Q and signer probabilities p_M, and p_X. Sample Python code to perform these calculations for any value of N is provided in Appendix E.1.

4.9 Risk Analysis

The model introduced in the prior section is dependent on the ability to estimate the actual probability of an individual signer being under the control of an attacker (p_M) or being unavailable (p_X). In reality, providing accurate estimates for these probabilities is challenging. These are rare events so data is limited. Section 4.11 suggests a method of incorporating assumptions about the process and observed data as a means of approximating these probabilities.

Another approach is to look at relative changes in risk. Starting with an approximate value for the unknown probabilities, the relative scale of the risk with a single singer and with multiple signers can be compared. This method shows the relative impact of multiple

signers. In fact, there may be cases where an attacker has already compromised a system, but had not yet made their presence known. In such a case, the real value of P_M would be higher than any estimate based on actual data.

Start with the assumption that $p_M \approx p_x$ and that both are unlikely events on the order of 3.4×10^{-6} as would be the expected value of a single error event in a six-sigma process. With these values, consider the case of a single signer, or specifically a case where $N = 1$ and $Q = 1$ to find that $P_C(N = 1, Q = 1) = P_D(N = 1, Q = 1) = 3.4 \times 10^{-6}$

When comparing probabilities of rare events and such small numbers, it is useful to focus on the relative order of magnitude of each result by taking the negative log of the net probability. In this case $-log[P_C(N = 1, Q = 1)] = 5.47$. This gives us a baseline probability of an undesired event where an attacker is able to deliver a modified update resulting in the client system becoming compromised.

Next, consider what happens if the system were designed to use 3 signers with a quorum of 2. Using the model introduced in the prior section, the result is $-log[P_C(N = 3, Q = 2)] = 10.46$. The significance of that result is that by using 3 signers and a quorum of 2 the probability of the recipient of a software update accepting a malicious update package has been reduced by about 5 orders of magnitude. This is achieved without changing the nature of the individual signers. The probability of the update being deferred due to singers being unavailable similarly goes down by 5 orders of magnitude.

Other combinations with 5 or 7 signers are shown in Table 4.2. This shows that adding 2 more signers ($N = 5$ and $N = 7$) and choosing a quorum that is a simple majority of signers reduces the probability of the recipient being compromised by an additional 5 orders of magnitude.

The comparative method can also provide guidance when a designer anticipates a significant difference between the probability of a signer being compromised vs. the probability that it will be offline. Choose values for p_M and p_X that represent the relative difference in likelihood of each event. For example, consider a case where with exceptionally

Table 4.2: Comparing results for multiple signers, $N = 3$, 5, and 7.

N	Q	$-log[P_c(N,Q)]$
3	2	10.46
5	3	15.41
7	4	20.33

Table 4.3: Comparing results with $N = 5$ and values of Q.

N	Q	$-log[P_C(N,Q)]$	$-log[P_D(N,Q)]$
5	1	6.30	25.0
5	2	13.0	19.3
5	3	20.0	14.0
5	4	27.3	9.00
5	5	35.0	4.30

good protection of our signing systems such that the probability of compromise is very low yet there may be occasional failures in the HSM used to protect the signing operation, such that the probability of a signer being unavailable is higher. Choose values that reflect this relative difference, such as $p_M = 1 \times 10^{-7}$ and $p_X = 1 \times 10^{-5}$. With these parameters and $N = 5$ the relative probabilities for an update being compromised vs. deferred, as shown in Table 4.3.

In order to minimize the probability that a recipient will accept malicious code (minimize P_C), it is desirable to set the minimum acceptable number of signatures (Q) to 5. However, this increases the risk that if any one signer is unavailable the recipient will not be able to validate the update so it will be deferred (P_D is maximum). Conversely, you could choose a lower value of Q to minimize the probability of a deferred update.

4.10 Unreliable Signers

The process of applying a digital signature on a software update is typically a highly controlled operation. However, there are some cases where signing code may be delegated to a large collection of less reliable signers. For example, an open source software package may be developed by a community of volunteers with a release process that is dependent

Table 4.4: Unreliable signers with a simple majority quorum.

N	Q	$-log[P_C(N,Q)]$	$-log[P_D(N,Q)]$
10	6	3.83	1.48
20	11	6.15	2.59
30	16	8.44	3.64
40	21	10.7	4.66
50	26	13.0	5.68

on a collection of developers who may be vulnerable to influence by malicious actors or be unavailable during the release process. Under these conditions it is reasonable to assume much higher values for p_M and p_X.

Consider an case where the probability of a signer becoming malicious is as high as $p_M = 0.10$ and the probability of a signer being revoked or unavailable during the deploy process is $p_X = 0.20$. Apply the risk model with the quorum set to a simple majority to evaluate the probability of the resulting system being compromised. The resulting probabilities are shown in Table 4.4

The risk model shows that with 10 signers, the probability of accepting a compromised update is on the order of $10^{-3.83}$ or about 1 in 10,000. With 20 signers, this is improved to $10^{-6.15}$ which is comparable to the single event failure rate for a six-sigma process. With 50 signers it can be reduced to $10^{-13.0}$. This last result can be achieved even when it is expected that there will be 5 signers that are malicious and 10 that are unavailable. With higher values of N, even higher levels of confidence can be achieved. This shows that under the assumptions used in this model, an arbitrarily high level of confidence (or conversely, arbitrarily low level of risk) can be achieved, even when the individual signers are unreliable.

4.11 Bayesian Estimation and Incorporation of Expert Judgment

In Section 4.9 hypothetical values were used for the underlying probabilities p_M, and p_X based on the assumption that $1 >> p_M + p_X$. In fact, the occurrence of a signer actually being *malicious* or *unavailable* may be so rare that there may be no observed occurrences

of these events. However, the actual probability of these events is non-zero. It may be useful to consider a Bayesian approach and assume that the uncertainty regarding these values may be expressed via a joint prior distribution. A natural prior distribution for p_M, p_X is the Dirichlet distribution as shown in Equation 4.6, defined over the simplex $\{p_M, p_X | p_i > 0, i = M, X, p_M + p_X < 1\}$ with parameters β, α_1, α_2, $\alpha_3 > 0$.

Equation 4.6 is a reparameterization of the original presented in Wilks (1962) to facilitate ease of definition of the parameters in terms of best guess values and variances such as $\alpha_1 = E[p_M]$, $\alpha_2 = E[p_X]$, and $\alpha_3 = E[p_T]$ (where $p_T = 1 - p_M - p_X$) and $VAR[p_i], i = M, X, T$ is a function of β alone (Ramsey, 1972).

$$P(p_M, p_X | \boldsymbol{\alpha}, \beta) = \frac{\Gamma(\beta)}{\Pi_{i=1}^{3} \Gamma(\beta \alpha_i)} (p_M)^{\beta\alpha_1 - 1} (p_X)^{\beta\alpha_2 - 1} (1 - p_M - p_X)^{\beta\alpha_3 - 1} \tag{4.6}$$

Following Mazzuchi and Soyer (1993), specify $\alpha_1 = \hat{p_M}$, $\alpha_2 = \hat{p_X}$, $\alpha_3 = 1 - \hat{p_M} - \hat{p_X}$ and let β represent a level of confidence in these initial values. A value of $\beta < 1$ indicates relatively low confidence in these initial values. β is sometimes referred to as the "prior sample size" associated with the estimates of $\hat{p_M}$ and $\hat{p_X}$. Mazzuchi and Soyer (1993) also note that other forms of the Dirichlet distribution may be used to preserve notions of ordering of the values. This approach can be used to enforce a constraint such as $1 > p_M > p_X > 0$.

Once defined, the prior distribution can be used for the risk analysis presented in Section 4.9 by using point or interval estimates from the prior or predictive outcomes using the predictive distribution as shown in 4.7. Note that for Equation 4.4 and Equation 4.6 used in 4.7, the resulting distribution has a closed form.

$$\iint p(n_T, n_M, n_X | p_M, p_X) P(p_M, p_X | \alpha, \beta) \, dp_M \, dp_X \tag{4.7}$$

Another advantage of the Bayesian approach if that observed values of n_T, n_M, and n_X (with $n_T + n_M + n_X = N$) are obtained, these may be used to update the prior distribution

via Bayes Rule to obtain a posterior distribution. In the case of the prior form given in 4.6 the resulting posterior distribution is a Dirichlet distribution with parameters $\beta^* = \beta + N$, $\alpha_1^* = \frac{\beta\alpha_1 + n_M}{\beta+N}$, $\alpha_2^* = \frac{\beta\alpha_2 + n_X}{\beta+N}$, $\alpha_3^* = \frac{\beta\alpha_3 + n_T}{\beta+N}$. This process may be repeated as more data is gathered.

4.12 Benefit-Cost Ratio

The expected value for the cost of a compromise can be calculated as the estimated cost of recovery (recovery after an event occurs) multiplied by the probability that the event will occur. The value of $P_C(N,Q)$ represents the probability of compromise for a single client system during a single update event. Section 4.8 showed that as more signers are added (i.e as N is increased), the value of P_C decreases. This suggests that for all $N > 1$ the expected value of the cost of compromise will also decrease. However, this decreasing cost is offset by the cost of the infrastructure, systems, and processes needed to configure and support N signers. The relative benefit of adding incremental signers can be approximated using a Benefit-Cost Ratio (BCR) calculation after defining some additional terms.

Recall that $P_D(N,Q)$ is the probability that a single update will be deferred or delayed. This can leave a client system vulnerable to specific attacks against known system flaws that may have otherwise been corrected if the update had been accepted. The impact of a deferred update comes in the form of a potential compromise. Let $p_{\bar{cd}}$ represent the probability that a malicious attacker will be successful in converting a deferred update into a compromise. We can then combine terms to form $p_{\bar{cd}}P_D(N,Q)$ which represents the probability of an update being deferred and converted to a compromise.

Let C_O represent the estimated cost to recover a single system that has become compromised. This cost may include real expenses such as recovery from a system failure, loss of life, and system down-time. The set $\mathbb{E}$ may be used to represent the list of all expected updates over the life of a client system of which there are L independent instances of client systems which must all receive the same update sequence. With these terms, the operational

risk R_O may be estimated as a function of N and Q as shown in Equation 4.8. This expression accounts for all update events (all elements in $\mathbb{E}$), the number of client systems in the field (L) and the cost for a single system compromise (C_c). This expression assumes that multiple client systems may be compromised, with an additional cost of C_c for each compromise event.

$$R_O(N,Q) = LC_c \sum_{\mathbb{E}} (P_C(N,Q) + p_{\bar{cd}} P_D(N,Q)) \tag{4.8}$$

Recognizing that the terms inside the summation may not change over the life of the device, the sum may be replaced by multiplication by the number of elements in the set of update events, $|\mathbb{E}|$, as shown in Equation 4.9.

$$R_O(N,Q) = LC_c|\mathbb{E}|\left[P_C(N,Q) + p_{\bar{cd}} P_D(N,Q)\right] \tag{4.9}$$

Section 4.9 showed that in cases where p_M and p_X are uncertain, it is reasonable to set Q to represent a simple majority where $Q = \lceil N/2 \rceil$ (i.e. the value of $N/2$ rounded up to the nearest whole number). Using this value for Q we can define R_O in terms of N as in Equation 4.9. With this expression, it is possible to calculate the benefit of using N signers instead of a single signer as $R_O(1) - R_O(N)$. This will be a positive value for all $N > 1$.

$$R_O(N) = LC_c|\mathbb{E}|\left[P_C(N, \lceil N/2 \rceil) + p_{\bar{cd}} P_D(N, \lceil N/2 \rceil)\right] \tag{4.10}$$

The cost for establishing and operating N independent digital signature functions can be approximated using C_E to represent the average cost per-signature to operate a signing system and C_F to represent the average fixed set-up cost per-signer. The total cost to operate N signers can be represented as $C_O(N)$ as shown in Equation 4.11. The increased cost for a system with N signers is higher than the cost of a system with a single signer by an amount equal to $C_O(N) - C_O(1)$.

$$C_O(N) = LN\left(|\mathbb{E}|C_E + C_F\right) \tag{4.11}$$

Bringing the relative risk ($R_O(N)$) and cost (C)(N) together, the net BCR can be defined as the benefit (in terms of reduction of risk) divided by the cost (in terms of operational cost) for N signers when compared against a single signer as shown in Equation 4.12. In this expression, the numerator represents the magnitude of the reduction in expected value of risk comparing a system with 1 and an alternate system with N signers. The denominator accounts for the incremental cost of the additional signing systems.

$$\text{BCR} = \frac{R_O(1) - R_O(N)}{C_O(N) - C_O(1)} \tag{4.12}$$

Substituting Equation 4.10 for $R_O(N)$ and Equation 4.11 for $C_O(N)$ provides Equation 4.13.

$$\text{BCR} = \frac{LC_c|\mathbb{E}|\left(P_C(1,1) + p_{\bar{cd}}P_D(1,1)\right) - LC_c|\mathbb{E}|\left(P_C(N,\lceil N/2\rceil) + p_{\bar{cd}}P_D(N,\lceil N/2\rceil)\right)}{LN\left(|\mathbb{E}|C_E + C_F\right) - L\left(|\mathbb{E}|C_E + C_F\right)} \tag{4.13}$$

Simplifying and combining terms produces the expression shown in Equation 4.14.

$$\text{BCR} = \frac{C_c|\mathbb{E}|\left[P_C(1,1) - P_C(N,\lceil N/2\rceil) + p_{\bar{cd}}P_D(1,1) - p_{\bar{cd}}P_D(N,\lceil N/2\rceil)\right]}{\left(|\mathbb{E}|C_E + C_F\right)(N-1)} \tag{4.14}$$

Section 4.9 showed that the use of multiple signers to protect an update package can significantly reduce the probability of a system compromise. The expression for BCR in Equation 4.14 provides a means for a system designer to use the multinational model to estimate the benefit of using N signers in a system. A system design with BCR > 1 indicates that the reduction in risk provided by adoption of two or more digital signatures outweighs the cost of configuring and operating the additional $N-1$ signing systems.

4.13 Chapter Summary

Several software update frameworks and digital signature structures can be used with multiple, concatenated digital signatures. In some cases, such as TUF, high-level guidance is provided to encourage the adoption of two or more signatures for critical roles. the number of client Without a model to show the relative benefit of multiple signatures, it is very difficult for system designers and operators to justify the high cost of creating and managing multiple signers for individual roles.

This chapter has explored the nature of digital signing systems and classified failures such that a software or parameter update mechanism may be compromised by an attacker or may become unavailable due to revocation or system failures. A probabilistic model was built to show the relative impact of multiple signers along with a method for updating the probability estimates with new information over time. A BCR calculation method was introduced to evaluate the relative benefit gained by adopting two or more signers.

Chapter 5: TUF with SHBS

The adoption of post-quantum algorithms for secure systems design is required due to continued advances in the development of quantum computing technology. The combination of approved SHBS algorithms with a managed software update process such as TUF form an attractive solution for post-quantum software update in critical infrastructure systems. The features of TUF may resolve some of the specific system design challenges introduced when adopting SHBS.

TUF was developed to address a variety of vulnerabilities in the software update supply-chain. It introduces a structured approach for managing and replacing trusted keys needed to validate future software update packages. When used with SHBS, this key management mechanism can be used to replace signing keys that have reached the limit on the number of signatures that they can produce. The same mechanism can also be used to remove and replace keys that may have been compromised or become unavailable due to a a state management failure.

These features come with the added cost of increased size in every digital signature, management of additional secure infrastructure, increased demand for secure storage, and higher computational effort needed to validate signatures. A simulation model was developed to evaluate the performance impact of SHBS in a basic update scenario, without TUF and then introduce a simulation method that can predict the performance impact of TUF with SHBS. Each of these use cases is then compared with respect to the storage, network usage, and computational cost of conventional public key methods currently in use. An optimization method for selecting SHBS parameters based on system requirements is also introduced.

This chapter is organized as follows:

1. SHBS Tutorial: Summary of the unique properties of XMSS and LMS algorithms which are the two approved SHBS methods endorsed by NIST, including benchmark

data that spans all of the approved parameter sets.

2. TUF Tutorial: This section provides details for the TUF approach to software updates and introduces a classification scheme for software update types.

3. Realistic Update Sequences: Introduces two real update applications and lists the update frequency for each with a forecast that spans the expected lifetime of each device.

4. Impact of SHBS with TUF: Describes the simulation model used to account for key rollover in TUF when used with SHBS and simulation results.

5. Chapter Summary: Summary of key insights and recommendations for critical infrastructure applications that require post-quantum security.

The main contribution of this chapter is to show that the key rollover feature integrated in TUF can be used to mitigate the limited lifetime of the approved SHBS algorithms. A simulation model is used with realistic data from deployed systems to show the impact of this combination on system performance. Finally, an optimization technique is proposed to automate the process of parameter selection for future applications.

5.1 Tutorial on SHBS

A digital signature is a method of proving the integrity and authenticity of a digital file. Most digital signatures currently in use rely on an asymmetric cryptographic method where you have a signing key that is used to create a signature and a public key that is needed for validation. If the recipient of a file is confident that they have the correct public key corresponding to a trusted sender, and they believe that the sender's key has not been compromised, then the recipient can trust that any content with a valid digital signature represents an authentic file that has not been modified or replaced in transit.

In order to describe the nuance of SHBS, it is helpful to re-write the representation of a generic signature algorithm that was introduced in Equation 4.1. Equation 5.1 describes a signature function where signature S is calculated from message M using signing key K_s using the signing function f_s. Validation is achieved by performing a complimentary calculation using the message, the signature, and a matched public key K_p as shown in Equation 5.2. The validation function f_v produces a binary result indicating that the signature S is valid for message M under public key K_p. This process requires that the key pair K_s, K_p be uniquely paired under the assumptions in the construction of f_s and f_v.

$$S = f_s(M, K_s) \tag{5.1}$$

$$f_v(M, S, K_p) \in \{0, 1\} \tag{5.2}$$

It is important to note that in this general form, the secret key K_s is stateless. This means that it can be used to sign an unlimited number of messages and there is no need to keep track of the history of signing operations. In fact, the secret key can be safely backed-up to secure storage and retrieved to recover the signing operation in the event that a secure singing system were to fail or become unavailable. Secure methods for key splitting and secure backup are commonly used to protect against this type of loss.

A hash-based digital signature is an algorithm that achieves these functions by using a secure one-way hash function as the core operation. This can be represented by including the state of the signer as X_t in the signature as shown in Equation 5.3. Each time the signing operation is performed, X_t is updated to reflect the new system state.

$$S_t, X_{t+1} = f_s(M, \hat{K}_s, X_t) \tag{5.3}$$

The dependency on state has two significant implications when using a SHBS algorithm. First, the secret key $\hat{K}_s$ has a limited lifetime. There is a maximum number of signatures that can be produced with this key. Once that limit is reached, the key must be discarded and replaced. This is fundamentally different from conventional public key algorithms. Second, the signing function must keep track of the key state, which is represented as X_t in 5.3. If the

signing algorithm were to sign two or more messages with the same state value (the same X_t), then any attacker who observes both signatures can extract enough information to forge future signatures with the same key.

These properties of all SHBS algorithms impose requirements for the signing process. NIST-Pub 800-208 requires that these algorithms are only approved for use when hardware protection is used to maintain and protect the system state for the signing operation. The reasoning is that any software-only implementation is inherently vulnerable to an attack where the system state may be replayed or reset to an earlier value. The use of a physically secure HSM, when properly implemented, can reduce this risk. An example of the type of hardware required is a special register that always increments in value every time it is read. By combining the read and increment into a single event, this type of special-purpose register can increase assurance that the hardware platform can not be tricked into repeating a former state with the same secret key.

The Stateful nature of these algorithms also makes it more difficult to ensure availability of access to the secret key. When a conventional signature algorithm is used, it is common to maintain one or more backup of the secret key in a separate hardware module or saved as key shares that must be combined to reconstruct the original key. In the case of SHBS, these techniques can not guarantee that a reconstructed key would not repeat the state of a former instance of the same key.

This property also introduces a new type of threat that system designers must consider when evaluating the security of a platform. With a conventional public key algorithm, it is sufficient to protect the secrecy of the signing key. When a SHBS is used, the system architecture must also ensure that the system state can never be reset to an earlier value. This means that if at any point there is uncertainty about the current state of a secret key, then that key must be revoked and treated as untrusted. This requirement must be enforced in addition to all of the other protections needed to maintain the integrity of a conventional digital signature system.

5.1.1 SHBS Benchmark Data

In order to evaluate the impact of introducing SHBS algorithms, it is valuable to have benchmark data that compares each of the parameter options in a consistent way. Table A.1 shows metrics collected for the XMSS algorithm and Table B.1 has the same parameters collected for the LMS algorithm. This data was produced using the open source `libOQS` library (Stebila and Mosca, 2017). Each row represents metrics associated with a unique SHBS parameter set. Each table shows results for the full set of parameter options available in the `libOQS` library.

For each parameter set the total number of signatures that can be produced is included in the table. The range of these values is quite large, so the log base-2 is shown. Also provided is the length of the signature (in KB), public key (in bytes), and private key (in bytes) for each parameter option. These values are used in the simulation model to compare the impact of each parameter set on an update scenario.

The last column shows the approximate number computational cycles required to validate a signature, measured as millions of cycles (M-Cycles). This data was collected using an x86 virtual machine. The cycle count is approximate as the actual measurement was process time which was then multiplied by the CPU frequency reported by the operating system. An average of 1000 time measurements was used to calculate the approximate number of cycles for each validation. The same method was used to calculate the computational effort to validate ECDSA P-256 signatures with the results shown in Table C.1. Note that for ECDSA, there is no limit on the maximum number of signatures, so no value is shown.

5.1.2 State Management

The requirement for state management introduces several special requirements on the use of SHBS. The NIST approval of these algorithms specifically requires that secure hardware, such as a dedicated HSM be used to protect the signing keys and the signing process. Secure storage and a protected computing environment may not be enough. Advanced attacks

such as fault injection through power glitching or external radiation could potentially cause a conventional microprocessor to repeat an operation or fail when writing a state update. Special hardware registers that increment a counter every time the current value is read are recommended as a method of minimizing the impact of this type of attack.

When using conventional public-key algorithms to sign secure software updates, it is possible to create a secure backup copy of the signing key to be used in the event that the active copy becomes unavailable due to a system failure. For example, a wide variety of methods have been developed to break stateless keys into secure key shares such that they can be re-assembled if required. This backup and recover process is not possible when SHBS are used. One option is to use a multi-tree structure where effectively one SHBS key is used to sign a collection of subordinate keys. These subordinate keys can then be saved in their own secure hardware with state management and used in the event that the primary key becomes unavailable.

A failure of state management introduces one way in which a signing key may become unavailable. For example, it is possible to have a situation where the key itself remains protected and available in secure storage, but the state management mechanism is unavailable or suspected of failure. If this were to happen, in order to maintain integrity of the system, it is necessary to consider the key to be compromised and no longer available to produce new signatures. Note that in this case past signatures produced by the key can continue to be trusted, provided that key itself has not been disclosed and the signing system is certain that state management was working correctly for all past issued certificates.

5.1.3 Impact of Parameter Selection

In addition to the requirement for managing the state of a signer, SHBS algorithms also produce larger digital signatures and require more computational resources than conventional signatures. Table 5.1 compares some key metrics between the conventional ECDSA using the popular P-256 curve with the range of options available for XMSS and LMS, the two

	ECDSA P-256	XMSS		LMS	
		Min	Max	Min	Max
Number of Signatures	NA	2^{10}	2^{60}	2^{5}	2^{40}
Signature Size (KB)	0.06	1.46	27.04	1.27	9.11
Public Key Size (B)	65	68	132	60	60
Validation Time (M-Cycles)	0.23	2.14	54.67	0.35	28.17

Table 5.1: Comparison of ECDSA with XMSS and LMS

SHBS algorithms introduced by NIST. A range of values is shown for each of the SHBS algorithms due to the variety of different parameter options available for each algorithm.

On a general purpose computing platform the differences among the various parameter sets may have a negligible impact on the overall system performance and security. On a system with a fast network, vast storage, and significantly underutilized computational capacity coupled with relatively infrequent software updates, even a highly inefficient set of parameters will have no noticeable impact on the overall system. The same is not true for highly optimized embedded systems. In some cases, these systems are designed to operate on the edge of a key parameter such as battery life or system storage. Some systems operate in extreme environments where the additional computational, communication, or storage requirements imposed by some parameter options would cause a previously viable system to fall short of the design goals. It is therefore valuable to consider the trade space introduced by the SHBS parameter sets.

5.1.4 Number of Signatures

A core property of any SHBS algorithm is that a signing key can only be used a limited number of times. This means that each key can only sign a limited number of messages. This has two impacts. First, the system state must be maintained to keep track of which components of the private key have been used and which ones are still available. Second, the overall system must account for the fact that each signing key is a limited resource that must

be replaced when it reaches the maximum number of signatures allowed for the selected SHBS parameters.

5.1.5 Public Key Size

Public key size impacts the storage required and may impact communication bandwidth in an event where a new public key must be distributed. In the most basic software update system, a single public key is installed during manufacturing and it is used to validate updates over the lifetime of the device, so the size of the public key only impacts the internal storage required. In a system where new public keys can be distributed to remote systems, the size of each key will impact the network bandwidth needed to distribute new keys.

5.1.6 Signature Size

Every secure message must be accompanied by at least one digital signature to validate the integrity and authenticity of the message content. Large digital signatures require additional network resources to distribute and store signed content.

5.1.7 Computational Cost of Validation

Many embedded systems have a limited budget of compute resources that can be allocated to the software update process. The computational effort needed to validate software update signatures may interfere with critical real-time control system functions, or require that they be scheduled during limited periods allocated for maintenance functions. In the case of a system that is battery operated, computation required to validate an update may have a direct impact on the battery life of the system.

5.1.8 Asymmetric Nature of the Update Process

The software distribution process is an inherently asymmetric process. There is typically one or a few points where new software is tested and validated for distribution while there

are many isolated client systems that will receive and validate those updates. In this type of system, it is efficient to place operations that require high computational load on the side of the sender. This structure aligns well with the nature of hash-based signatures.

Keypair generation, which typically requires significant compute resources, is concentrated at the few signature generation locations. The signing operation, where careful state management and special hardware systems are required, is similarly required only on the sender side. The client systems are not impacted at all by the requirement for state management or by the size of the secret key used for signing. Client systems are only impacted by the size of the signature itself, the size of the public key (or keys) needed for validation, and the computational impact of the validation operation.

5.2 Tutorial on TUF

A typical secure update process involves applying a digital signature to the hash of an update package. With this approach, each recipient is provided with a trusted copy of the public key which is used to validate signatures on all future update packages. When a new update is received, the public key is used to check the integrity of the update file. This method allows the recipient to test that the update package came from the authorized sender and that the content of the update package had not been modified.

This approach only works when several critical assumptions are true. The method assumes that the single private key used to sign updates has not been compromised and that the system that stores that key is only accessible to authorized users who follow a trustworthy software release process. It also assumes that the newly received update is actually new and not a copy of an older package which may contain a known vulnerability or content that is not compatible with the current state of the recipient's system.

TUF was created to provide a reliable framework that brings clarity to these assumptions and addresses several common vulnerabilities in the software update process. In particular, it was intended to protect against the threat that a single key compromise could put an entire

deployment system at risk. TUF achieves these goals by distributing the trust model for an update package among several distinct *roles*. Each role in TUF provides protection against specific types of threats. This is accomplished by applying one or more digital signatures at various stages in the release process. A specific instance of a TUF update system will be referred to as an update repository, sometimes referred to as a "repo" for software systems.

There are four roles in TUF:

1. Root: Generates a signed list that includes all roles (including new Root roles) with their corresponding public keys and policy requirements including the required quorum of signatures. A signed message from the Root role may be used to update or replace the public key associated with any role in the framework.

2. Target: Produces a signed manifest with the hash of each software package that is included in an update. This role is responsible for directly processing the files associated with a new software update. Each new file is vetted to ensure it complies with all policy requirements, such as successfully passing all development tests and quality checks. There may be multiple Target roles in one update repository. One Target role must sign every update file.

3. Release: Creates a signed list of all currently valid Targets. The Release role allows for a TUF repository to distribute many different types of software packages which may contain different combinations of files. Each updated file is signed by a Target role, and then a collection of targets will be signed by the Release role.

4. Timestamp: Identifies the current version of all currently valid releases. The Timestamp role provides a trusted source of time that client systems can use to know the proper sequence in which updates must be applied. This prevents a "roll-back" attack where an attacker may deliver older code with a valid signature. The Timestamp role also allows client systems to become aware if they are being blocked from the update infrastructure. By signing and distributing a new Timestamp file at a regular

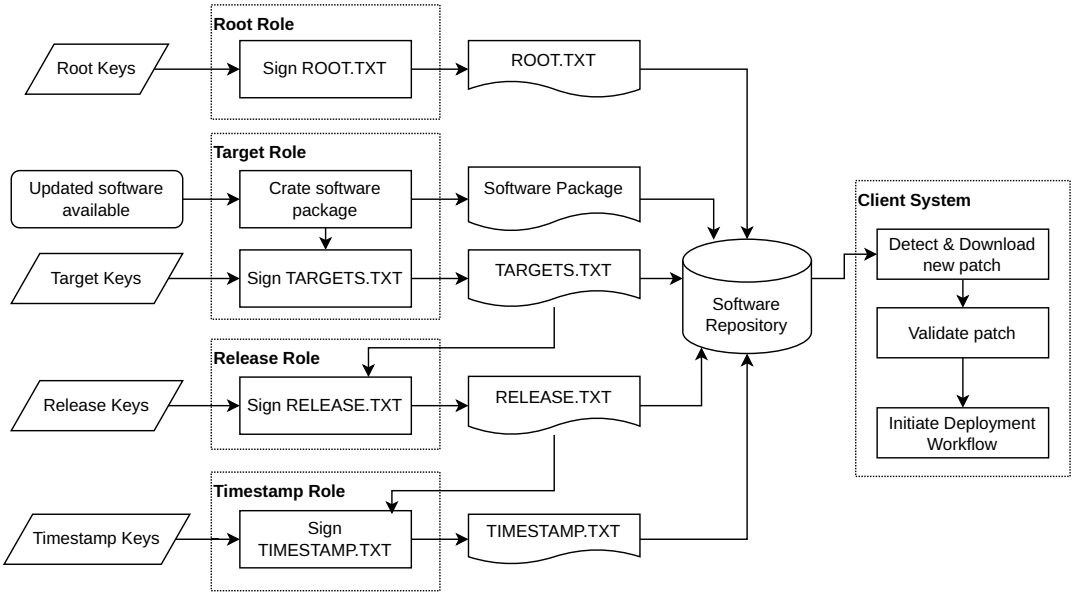

Figure 5.1: Simplified view of the TUF process flow.

interval (such as once per-day), client systems can be configured to expect to see new Timestamp signatures at a constant rate. If one or more signed timestamp is not delivered, a client system may be able to alert an operator or take some other protective action.

A simplified view of the TUF process is shown in Figure 5.1. The four roles are shown, each possessing one or more private keys which are used to execute a digital signature operation. The Root role produces a signed file (`Root.txt`) which contains a signed list of all public keys currently in use. This file can also introduce new public keys which need to be distributed to client systems. This feature will be used in Section 5.2.1 to describe how SHBS keys can be replaced when they reach the end of their useful life.

Continuing with the simplified process in Figure 5.1, a new software package is introduced to the Target role. This role produces a file containing a digital hash (a binary representation of the entire update file) and a signature which can be used to prove that the update file has not been modified. This signed file is distributed as `TARGETS.TXT` which is sent to client systems in addition to being forwarded to the Release role. In the context

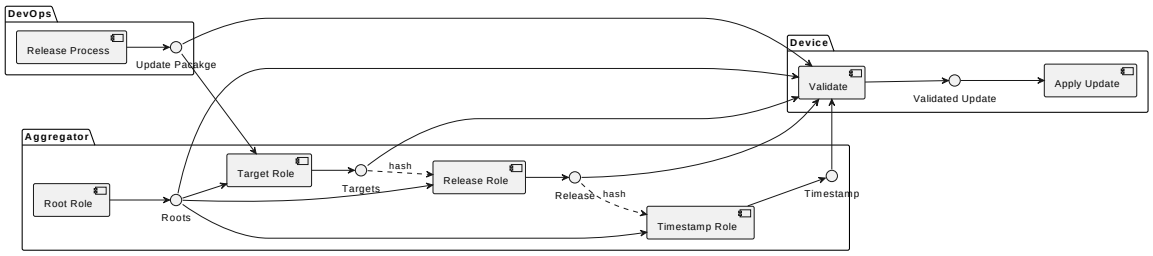

Figure 5.2: Release workflow integrated with TUF.

of TUF, a "release" is a collection of updates that are intended to be applied at the same time. The Release role may collect multiple `TARGETS.TXT` contributions and assemble them into manifest which is signed and distributed as `RELEASE.TXT`. The Timestamp role is intended to generate a signed message at regular intervals. This message contains the identity of the most recently produced `RELEASE.TXT` file and the current time. Client systems can use the resulting `TIMESTAMP.TXT` file to know if they have received and applied the latest update. Client systems should be configured to expect to receive updated and signed `TIMESTAMP.TXT` files on a regular basis. If a Timestamp file fails to arrive, it could mean that an attacker is trying to prevent the client system from obtaining the latest update. This could allow an attacker more time to analyze and execute an attack against a known vulnerability.

Figure 5.2 shows how TUF can be integrated into an overall software release workflow. This view shows the handoff of a candidate release package from the SecDevOps process into the Target role and to client systems. This transition must be a closely coordinated process that ensures that an attacker can not insert malicious code into this stage of the workflow. The Target role may be fully automated or it may require direct operator interaction. In either case, it is required that only authorized software that meets all quality checks and internal review is allowed to move through this distribution step.

5.2.1 Key-Pair Rollover

An important feature in the TUF model is the fact that the Root role can update or replace the public keys used by each of the other roles, including the Root role itself. In a case where a signing key has been compromised or is unavailable due to a system failure, this feature allows for all client systems to be securely updated with a new, trusted public key so that the signing authority of the compromised role can be restored.

In the case where Stateful signature algorithms are used, this feature allows for the replacement of a signing key that has exhausted all of the signatures that it is allowed to produce. The process for replacing a key is as follows:

1. Generate the new key pair
2. Securely deliver the new public key to the Root roles
3. Root roles generate a new ROOT.TXT file containing all currently valid public keys with their parameters, including the newly generated public key.
4. Concatenate the signatures from all Root roles into a new ROOT.TXT file.
5. Distribute the new ROOT.TXT file as part of the update package sent or downloaded by all client systems.
6. Client systems validate all of the Root role signatures. If a quorum of Root signatures are valid, the list of public keys in the file replaces the set of trusted public keys for the update process.

5.3 Software Update Sequence

While there are many unique reasons that can trigger a need for a software update. It is helpful for modeling purposes to classify three distinct update types. These will be classified as *planned* updates, *critical* updates, and *contingency* updates. Understanding the nature of

these classifications will be valuable when planning the number of updates that might be required during the lifetime of a real CI system once it is installed.

It is typically not necessary for every client system to install every update. System operators may choose to skip updates that are not likely to improve the system performance or features based on their specific use case. Historically, industrial systems are only be updated when there is a critical security fix that addresses a known vulnerability. Modern best practices strongly encourage that system operators do apply updates as soon as they become available. This ensures that all security patches and improvements are applied as soon as they are made available. For modeling purposes, take the view of a client system where all updates will be applied as soon as they are available.

5.3.1 Planned Updates

Modern computer systems often run a wide variety of software packages that are maintained by independent organizations. Software updates can be delivered from many different sources at various times and applied based on a local policy. This can produce a very complex environment with software packages that can interact with each other in unpredictable ways.

In the case of mission critical systems such as industrial controls, medical, or military equipment, it is common for the device vendor to provide a sequence of consolidated update packages. Each package may include contributions from individual vendors combined and tested to ensure that the combination works as intended on the target system.

When a new product is introduced, the frequency of these consolidated updates is typically higher than it is for a mature product. For example, a modern control system may provide quarterly updates for a period of time after the product launch. These updates may include a mix of security-relevant fixes, functional bug fixes, and new or improved features. This level of activity typically lasts for a period of 3 to 5 years. Once the product reaches a level of maturity, the frequency of planned updates may reduce to a lower level, such as annual for the remaining lifetime of the product.

5.3.2 Critical Updates

Events rarely proceed as planned over a time span as long as 10 years. There are situations where critical security flaws will be discovered in the underlying operating system, or a critical software component in the system that requires immediate attention. Rather waiting for the next planned quarterly or annual update package, a separate critical update package is created and delivered to system operators.

An example of this situation is the `Log4Shell` event, documented in `CVE-2021-44228`. This was a serious security flaw that was discovered in December, 2021. It impacted the Apache `Log4j` library which is found on a wide variety of embedded systems. On vulnerable systems, this flaw allowed an attacker to execute arbitrary code on the target system. Many device vendors immediately published advisory information on how to protect systems and limit the potential damage. However, there was significant pressure to provide a software update to fully eliminate the threat. Many device vendors created an unplanned critical update security patch to address this specific issue.

5.3.3 Contingency Updates

Over the long lifespan of many industrial devices, there are events that may impact the software deployment infrastructure. For example, a signing key used to approve and sign update packages may become unavailable due to a system failure, accident, or malicious action. It may also be discovered that a signing key has become compromised and made available to potentially malicious attackers. In both of these cases, a change is needed in the software distribution infrastructure to correct the problem and return to the normal update sequence.

When a simple update mechanism is used, such as a single key pair used to approve all updates, the ability to respond to this type of key loss or key compromise is very limited. If the public key used to verify the software was issued through a PKI system, then it may be possible to add a compromised or lost key to a CRL. This will effectively block client

systems that have access to the CRL from trusting the compromised key. If no PKI system is used, then there may be no automated way to notify remote systems that a code signing key has been compromised. In this situation is is common to revert to a less-secure manual update distribution. Unlike a typical software update, this manual update procedure will require changing the device trust store.

This type of infrastructure compromise is where a structured update framework like TUF and its derivatives is uniquely valuable. In the case of TUF, a key compromise can be corrected by issuing a new trust list through the Root role. Provided that a quorum of root signers are available to sign a new root update message, they can publish a revised trust list which can be automatically applied by client systems.

5.4 Realistic Update Sequences

The impact due to adoption of SHBS on an update process is dependent on the number of updates that must be delivered to client systems. While it is possible to support a large number of signatures of up to 2^{60} using approved XMSS and LMS parameter sets, these options come with a cost of a significant increase in signature size or computational load to validate signatures. In order to consider a realistic number of updates for a real critical infrastructure system, it is helpful to look at two actual deployments and estimate the lifetime number of updates for those systems. These examples can then serve as reference points when approximating the impact on comparable systems.

Consider two critical infrastructure solutions with very different update strategies. First, examine a commercially developed Ethernet switch which is frequently used in power generation and distribution networks. Then examine the AWS Bottlerocket project which delivers a real-time operating system that may be used in a variety of infrastructure systems.

5.4.1 Ethernet Switch

A modern Ethernet switch can support a wide variety of use cases including security-critical functions such as establishing network segmentation and filtering potentially malicious packets from a network. The long-term integrity of a switch is dependent on both the timeliness of software updates to respond to changing threats as well as the quality of those updates which must undergo a rigorous testing and validation procedure before they are made available to client systems.

A leading manufacturer has managed this balance by establishing a regular cadence of quarterly software updates for a product that was introduced to the market in 2019. Each quarterly update includes an accumulation of new features, bug fixes, and security updates that have been developed in the prior quarter. By choosing a regular release cadence, the manufacturer can schedule resources for the extensive testing and issue remediation cycles needed to fully qualify a new software build prior to releasing it to the market.

Early in the product life there was an increased period of rapid development with frequent software updates made available to include new features and to address bugs as well as security patches. Over time, the rate of updates has been reduced to the planned quarterly sequence. The company plans to maintain this frequency of updates for a period of active development which is expected to last for 5 years. After 5 years, the pace of development will be reduced and product is expected to transition to annual updates. The expected service life of the product is expected to be 10 years. However, past experience has shown that some customers may continue to operate the product for a much longer period, sometimes up to 25 years.

For modeling and simulation purposes, it will be assumed that quarterly updates will continue for a period of 25 years. This is a conservative estimate as the manufacturer expects to reduce the update frequency over time, so modeling quarterly updates allows for four times the number of planned updates in future years. Table 5.2 shows the sequence of updates per-year, with actual update counts shown for 2019 through 2024 and a conservative

2019	1	2024	4	2029	4	2034	4	2039	4
2020	6	2025	4	2030	4	2035	4	2040	4
2021	6	2026	4	2031	4	2036	4	2041	4
2022	4	2027	4	2032	4	2037	4	2042	4
2023	4	2028	4	2033	4	2038	4	2043	4

Table 5.2: Device A (Ethernet Switch) update sequence.

estimate for planned updates in future years. This sequence of updates will be referenced as "Device A" in subsequent analysis. The sequence assumes a total of 101 software updates over a span of 25 years.

5.4.2 Embedded Operating System

The Bottlerocket project is an open source embedded operating system that was released in 2020. This project is intended for use in a wide variety of IoT and industrial systems. Like many open-source projects, the code is maintained in a public repository that is available for developers to clone and copy at any time. Periodically, the project maintainers decide that there is enough new functionality in the code that they generate a formal release. Products that incorporate the Bottlerocket OS may make the released updates to the OS available to their end users through their individual software update mechanism. These packages may include updates to additional application layer software as well as security patches in each subsequent release.

For the purpose of analysis, the update frequency is based on the cadence of named releases in the Bottlerocket repository. Unlike the planned quarterly releases that are expected for the Ethernet switch example, the frequency of updates for Bottlerocket are more opportunistic. The short sequence of data for 2020 through 2024 (shown in Table 5.3) appears to follow a Poisson process with $\lambda = 18.4$. The Python numpy library's `random.poisson()` function was used to generate future sample values to represent the number of releases in future years.

This approach assumes that active development will continue on the Bottlerocket project

2020	20	2025	19	2030	24	2035	13	2040	17
2021	17	2026	10	2031	22	2036	27	2041	20
2022	16	2027	16	2032	29	2037	14	2042	13
2023	16	2028	9	2033	19	2038	17	2043	22
2024	23	2029	17	2034	13	2039	19	2044	29

Table 5.3: Device B (Bottlerocket OS) update sequence.

at the same sustained rate for a span of 25 years. This is highly unlikely as most open source projects eventually settle into a much lower frequency of updates. As with the Ethernet switch example, this can be considered to a conservative estimate for the future release requirements for this project. Table 5.3 shows the combined data (actual and calculated) for the Bottlerocket OS. This sequence of updates will be referenced as "Device B" in subsequent analysis. This sequence of updates results in a total of 461 updates.

5.5 Impact of SHBS on Basic Updates

A basic form of secure software update is to simply apply a single digital signature on each update package. Validation requires that client systems must be provided with a trusted copy of a public key for validating signatures. In this model, a single key pair is typically used for the life of the product.

For the two sample device types, it was estimated that there is a need for 101 updates during the life of Device A and 461 update for Device B. A review of Appendix A and B shows that there are many configuration options that can far exceed the required number of signatures. For an application where computational efficiency is an important consideration, then a good parameter choice is "LMS_SHA256_H10_W1" which can support $2^{10} = 1024$ signatures with a validation effort of 0.38 M cycles and a signature size of 8.64 KB. If minimizing the size of each digital signature is more critical, then a good option is "LMS_SHA256_H10_W8" which also supports 2^{10} signatures, but has a signature size of only 1.42 KB and a validation effort of 6.43 M cycles.

For comparison, the current practice of using ECDSA signatures with a 256 bit curve

	LMS_SHA256_H10_W1		LMS_SHA256_H10_W8		ECDSA P-256	
	Total Signatures (KB)	Validation Effort (M-Cycles)	Total Signatures (KB)	Validation Effort (M-Cycles)	Total Signatures (KB)	Validation Effort (M-Cycles)
Device A	873	38	143	649	6	23
Device B	3,983	175	655	2,964	30	106

Table 5.4: SHBS compared to ECDSA for the basic update scenario.

(as shown in Appendix C) has a signature length of just 0.06 KB and validation effort on the order of 0.23 M cycles. That means that switching to SHBS requires approximately $0.38/0.23 \approx 1.7$ times as much effort to validate each signature or a signature size that is $1.42/0.06 \approx 23.7$ times bigger than an ECDSA signature. A summary of these comparisons is shown in Table 5.4.

5.6 Relative Cost of Post-Quantum Software Updates

The TUF process requires that client systems receive many more signed files and public keys than a conventional secure update process. With the switch from conventional signature algorithms such as ECC to a post-quantum approved algorithm like LMS or XMSS, each of the signatures and public keys will grow in size. In addition, due to the limited number of signatures available for each key pair, it may be necessary to distribute new public keys frequently.

It is useful to consider the relative cost introduced by the transition to post-quantum algorithms when compared to the use of TUF with traditional signatures.

5.7 Simulation Model

The simulation model used in this work steps through the logical steps of the TUF protocol and accounts for the number of digital signatures that must be generated, the cumulative size of each update metadata file, and the relative compute resources required to validate each update package. These incremental calculations are then combined to come up with an

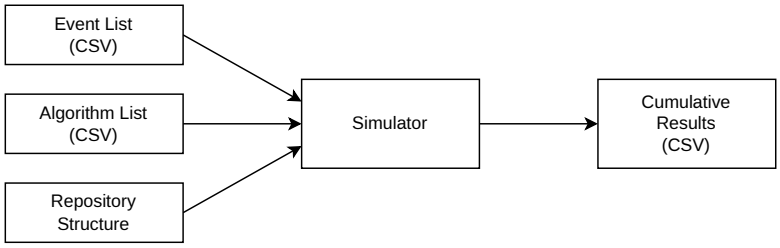

Figure 5.3: TUF simulation high-level description.

overall impact of a set of parameter choices over the expected lifetime of a deployed product. The simulation does not directly implement the TUF protocol or the signature methods but it accounts for the size of every signature and public key that must be distributed as well as the cumulative computational load required for a client system to validate signatures. Figure 5.4 shows the overall flow of the simulation.

Figure 5.3 shows a high-level view of the simulation process. The core simulator takes as input a list of algorithm types with a set of parameters for each algorithm. In the case of the SHBS signature algorithms, this file includes the a separate entry for each approved parameter set. Each entry includes the parameter set name, the maximum number of signatures, sizes of the signature and public key, and the computational effort for validation. A separate file contains a list of events. This list defines the specific date and time when each simulated update will occur. The simulator can also be configured with a repository type. This is where the number of signers is defined for each role in the TUF update process. The output of the simulation is a Comma Separated Value (CSV) file that contains cumulative results from the defined sequence. The simulator can be configured to iterate through all of the algorithms. For each algorithm in the list, it will run a simulation with the same algorithm assigned to all of the TUF roles and produce a cumulative output. The resulting CSV output file contains separate metrics for each algorithm.

Figure 5.4 shows a state machine diagram that represents the overall simulation process. The “rollover” blocks in this figure represent the process of updating the keypair for a role that has reached the maximum number of signatures allowed for the current parameter set.

When this happens, the simulator accounts for the process where the root role will produce a new file containing all of the public keys currently active in the system. The size of the Root signature is then accounted for as part of the cumulative size of all update signatures.

The simulation process accepts an "update profile" as input. In the case of the sample devices (Device A and Device B), this is the sequence of annual updates listed in Tables 5.2 and 5.3. The simulator assumes that a new Timestamp signature will be generated every day, during the life of the product. This happens whether there is a new update to be distributed or not. Client systems can download the latest signed Timestamp file at any time to find out if they are running the most recently released software. If the latest Timestamp file indicates that there is a newer update available, then the client system will download the latest file and validate the manifest using the Target public key. The Timestamp can also notify client systems if there is a new Root file available at any time.

Another type of event is a key compromise. This is a situation where the private key used to sign messages on behalf of one of the roles in the TUF deployment model becomes unavailable or untrusted and needs to be replaced. In a conventional PKI system, this situation would involve the revocation and replacement of a signing key. The TUF framework allows for this by issuing a new ROOT.TXT file, signed by the Root roles and made available for clients to download. Key compromise events can be modeled in the simulator as exceptions, along with additional updates triggered by a security flaw. In the simulator, these unscheduled events may be provided as a list of discrete, timed events. This allows for multiple simulation runs to be performed using an identical set of update activities. The list of events may be generated based on historical data or with a probabilistic model.

The simulation model accounts for the size of all digital signatures and public keys that must be distributed to clients. It also accumulates the net compute effort required by a single client to validate all of the signatures. The size of each public key and signature as well as the compute effort is calculated based on the signature algorithm parameter set that has been selected. The output of the simulation is a cumulative total of the communication cost

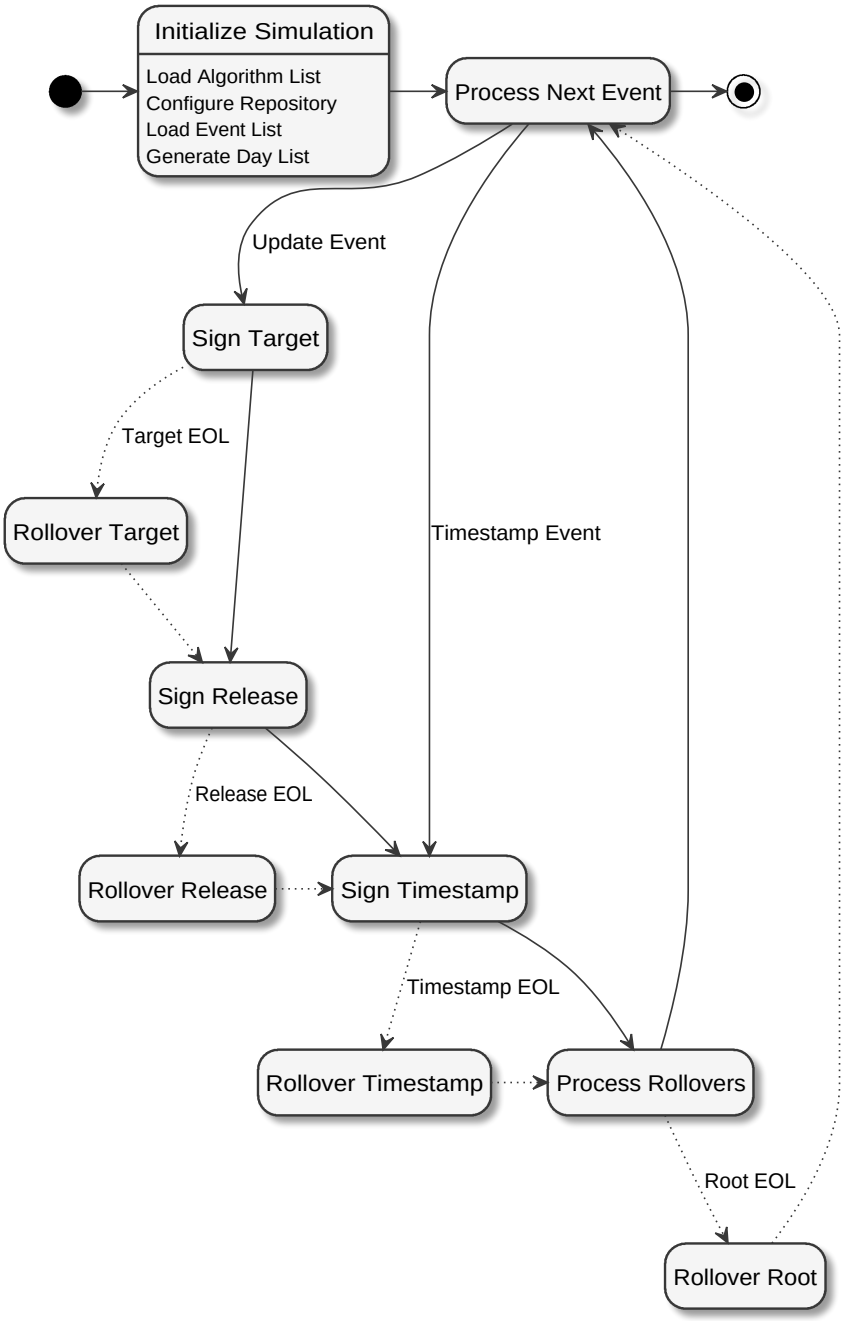

Figure 5.4: TUF simulation sequence diagram.

for distributing keys and signatures and the estimated compute effort required to validate signatures. The number of key replacements performed is also reported.

The simulation model does not account for the number of client systems that may be impacted by an update package. The bandwidth and compute impact reported are accurate only from the perspective of a single client device. The accounting further assumes that the client device downloads all of the updates that are made available by the software distribution system. The network bandwidth and compute resource estimates provided by the current simulation model represent a "worst case" scenario. It is important to recognize that in many real-world systems, client devices may skip some some update packages, so the actual impact on an individual device may be lower.

5.8 Simulation Results

Simulation results for Device A and Device B are included in Appendix D, Tables D.1, D.2, D.3, and D.4. These results show the cumulative size of all signatures and the total validation effort for each device using all available SHBS parameter sets. To provide a reference, the last entry for each device also shows the result of a simulation that uses ECDSA with a P256 curve. In producing these results, the same parameter set was used for all TUF roles in each simulation run.

For Device A, the minimum number of signatures is 9,711. This represents the case where no roles run out of signatures, so there is no rollover events during the life of the device. This is the case for the majority of the simulation runs. This means that even with daily timestamps being produced and a 25 year lifespan for the device, most SHBS parameter sets can support enough signatures such that they never need to be replaced. For the parameter sets where the total number of signatures is greater than 9,711, it indicates that the TUF process seamlessly replaced the impacted signer and securely deployed a new keypair, mitigating the SHBS limitation on the total number of signatures. The same effect can be seen for Device B, where the minimum number of signatures is 10,411.

Assume that some client systems will download and validate all (or the majority of) the signed files, then it may be desirable to look for the parameter options that minimize the total validation effort or total cumulative file size. In the case of Device A, the minimum validation occurs when the LMS_SHA256_H5_W1 parameter set is used, resulting in a cumulative validation effort of 3,486 M-Cycles. This is a surprising result because this parameter set only supports $2^5 = 32$ total signatures for each key-pair. During the life of the device, the timestamp key pair must be replaced 286 times. However, the LMS_SHA256_H5_W1 parameter set is substantially more efficient than any other configuration, so even with this overhead it is still the option with the minimal validation effort.

If the goal is to minimize the cumulative size of all signatures and public keys that need to be distributed to clients, the result is similar. In this case, the optimal choice is LMS_SHA256_H5_W8. This is another parameter option that only supports 32 total signatures and requires frequent rollover operations. However, due to the small size of the signatures, this parameter option is still superior than any other option. The results are the same for Device B with over 4 times as many updates over the life of the device. The same two parameter choices produce the optimal results.

For both devices, the optimal time is about 2.6 times larger for SHBS when compared to ECDSA. The minimum size option has a cumulative total that is about 21.0 times larger than ECDSA. These values are comparable to the basic update calculations presented in Section 5.5 which were 1.7 times and 23.7 times respectively. This means that by adding the resilience and key replacement capability of TUF, the relative cost of software update over the life of two representative industrial systems is comparable to the relative cost of doing a basic update with a single key pair.

Figures 5.5 and 5.6 show the simulation results for each device in a log-log scatter plot. In both figures, the point labeled as 'A' represents the computational effort and cumulative download size for signature overhead when ECDSA is used as the signature algorithm. Each black circle represents the results for one of the approved SHBS parameter sets. The points

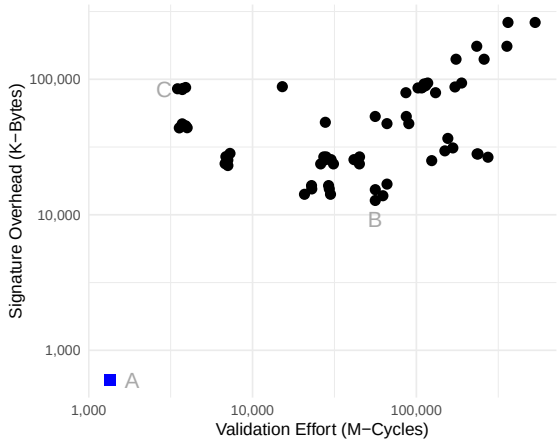

Figure 5.5: Device A with 1 Root signer.

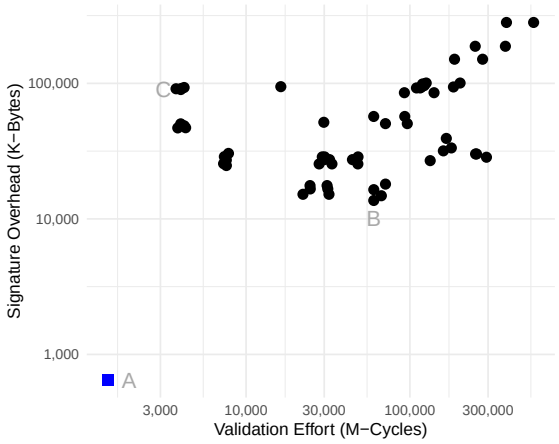

Figure 5.6: Device B with 1 Root signer.

labeled as 'B' represent the parameter option with the minimum signature overhead for each device. The points labeled as 'C' show the options with the minimum cumulative validation effort.

The two scatter plots are nearly identical. This is not surprising as the update sequences for the two devices being modeled are quite similar. For both devices, the collection of points between points 'B' and 'C' represent parameter options that offer a trade-off of minimum computational effort versus minimum download size. This sequence of points represent a Pareto Frontier, the set of the most efficient options available among the approved parameters. System designers may choose any of the parameter options along this frontier if they require a balance of minimum computational effort and download size.

5.9 Multiple Roots

As discussed in section 5.2, the TUF method allows for multiple singers in each of the four roles. Chapter 4 showed that the risk of system compromise can be dramatically reduced when multiple signers are available to mitigate the risk of key compromise or loss. Samuel et al. (2010) explain that the Root role is the most critical. A key compromise in this role would allow an attacker to replace all other roles with keys of their choice and deliver malicious updates at will. Therefore, it is useful to consider the impact of using multiple

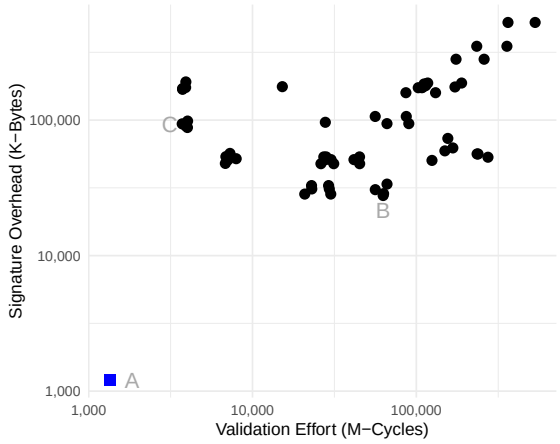

Figure 5.7: Device A with 5 Root signers.

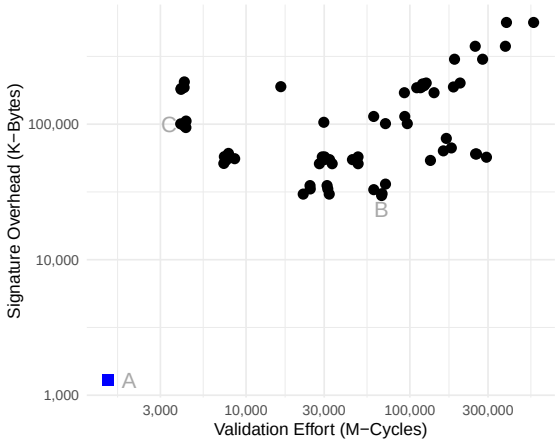

Figure 5.8: Device B with 5 Root signers.

signers to protect this critical role in the update framework.

The simulations for both device types were run with five signers in the Root role. The results are shown in Figures 5.7 for Device A and and 5.8 for Device B. Close comparison of these results will show that the points labeled as the minimum overhead selection (point 'B' in each graph) and the minimum computational effort (point 'C' in each graph) have moved when compared with the simulation with a single signer, shown in Figures 5.5 and 5.6. The differences can be seen more clearly in Table 5.5 which shows summary results comparing the performance with 1 and 5 Root signers for each device. Once again, the ECDSA results are included as a reference.

The data in Table 5.5 shows that the best performing SHBS parameter set changes for both devices when switching from a single Root role signer to 5 signers. This is due to the fact that with five signers, the performance impact of a root rollover is significantly increased. As a result, the best performing parameters have much larger number of signatures available. Looking up LMS_SHA_256_H10_W8 and LMS_SHA_256_H20_W2 in Table B.1, we see that they have 2^{10} and 2^{20} signatures, respectively. This change means that there will be no required root rollover events during the operational life of either device. Comparison of the performance metrics when switching from one signer to five shows that the impact of this change on download bandwidth is relatively small. In the case of Device A, the

Device	Root Signers	Parameter Set	Download Size (KB)	Validation Effort (M-Cycles)
A	1	ECDSA P-256	1,340	607
A	1	LMS_SHA256_H5_W8	56,137	12,756
A	1	LMS_SHA256_H5_W1	3,486	85,095
A	5	ECDSA P-256	1,341	1,212
A	5	LMS_SHA256_H10_W8	62,714	27,760
A	5	LMS_SHA256_H20_W2	3,722	93,811
B	1	ECDSA P-256	1,437	651
B	1	LMS_SHA256_H5_W8	60,181	13,675
B	1	LMS_SHA256_H5_W1	3,737	91,226
B	5	ECDSA P-256	1,438	1,302
B	5	LMS_SHA256_H10_W8	67,212	29,750
B	5	LMS_SHA256_H20_W2	3,991	100,570

Table 5.5: Comparison of results for 1 and 5 Root signers.

minimum cumulative download size changed from 3,486 KB to 3,722 KB which is a 6.8% increase. The minimum cumulative validation effort changed from 12,756 M-Cycles to 27,760 M-Cycles, a 118% increase. Similar changes are observed for Device B, indicating that this change is not unique to a specific update sequence.

5.10 Chapter Summary

This chapter has shown that SHBS can be effectively combined with TUF to create a practical update solution for critical infrastructure systems. This work supports the recommendation in NIST 800-208 that encourages deployers of systems that will be in the field for a long period of time to adopt SHBS as a post-quantum algorithm now. SHBS are an attractive post-quantum option because they rely on very few assumptions.

The specific combination of SHBS with TUF effectively mitigates some of the challenges associated with stateful signatures. Specifically, it eliminates concerns over the limited number of signatures available for each key pair since the TUF model provides a secure method for distributing replacement keys. Note that this feature does not eliminate the need for careful implementation of the signing systems to ensure that they securely maintain state.

However, there are generally just a few signing systems and they are centrally managed where more complex hardware and operational requirements can be effectively maintained. Client systems which are much more numerous and where cost and the operational environment is highly constrained do not have this requirement. The ability for the TUF method to securely distribute replacement keys to these devices after they are deployed in the field is a significant benefit when SHBS algorithms are used. These results were obtained using a simulation model which can accurately calculate the total number of signatures required over the lifetime of a product based on an assumed update profile. Sample data has been provided for two representative devices, an Ethernet switch and a real-time operating system. The results show that the application of TUF to these systems requires about 2.6 times as much computational effort to validate all signatures and 21 times cumulative file size to be downloaded when compared to ECDSA with a P256 curve.

Chapter 6: Automatic Parameter Selection

The results discussed in Chapter 5 were all produced using the same SHBS parameter set for every TUF role. The TUF design can also support deployments where each role has a distinct digital signature method. This may be an attractive option as each role in TUF has unique requirements. For example, the Timestamp role must sign many more messages than any other role while the Root must be protected more carefully than any of the others. This raises the question of how to select SHBS parameter options if different parameters are to be chosen for each role.

6.1 Optimization Method

The simulation results presented in Chapter 5 yielded the surprising insight that the optimal choice (when a single Root signers is used) is a parameter set that only supports 32 signatures. This is possible because signing keys that have reached the end of their useful life can be replaced in the field. The impact of these key rollover operations can be calculated using simulation, but the interaction between different roles is not intuitive or obvious. For example, Table 5.5 shows that in switching from one Root role signer to five, the best performing parameter set changed to a version with a much larger number of signatures. When considering unique parameters for each of the four roles in TUF, a change in one role can have unexpected results due to interactions with the other roles over the operational life of a product. Exhaustive testing is not practical. With 4 roles and 63 parameter options for each role, there are $4^{63} \approx 8.5 \times 10^{37}$ parameter combinations that could be tested.

One approach that can be effective for this type of optimization challenge where the choices are distinct categorical values is to use a genetic algorithm. A cost function can be defined that is a linear combination of overall system characteristics that are desired. For example, an optimal solution might be one that balances the priorities of low validation

execution time and small update file sizes. It is helpful to normalize these metrics so that they are on a comparable scale.

6.2 Cost Function

Equation 6.1 is a linear combination where M_v represents the total execution time for validation and M_s is the total size of all update files, both calculated using simulation under a candidate parameter set. The scale factors of 32,000 and 15,000,000 were chosen based on the maximum values seen in the suite of simulations run for a single parameter. These scale factors serve to balance out the relative scale of the validation time and cumulative update size values. The weighting factor w_v can be selected as a number between 0 and 1 that represents the relative significance of validation runtime on the desired system performance. When set to 1, the optimal choice will be selected to minimize validation runtime, when set to 0, the optimal choice will prefer smaller update packages.

$$C = w_v \frac{M_v}{32000} + (1 - w_v) \frac{M_s}{15000000} \tag{6.1}$$

6.3 Parameter Selection and Results

This cost function was tested along with the TUF simulation model and the Python PyGAD optimization algorithm. The PyGAD code is designed to maximize a "fitness" criteria rather than minimize cost, so fitness was defined as $F = \frac{1}{C}$. A set of 4 genes were defined, where each "gene" represented the choice of a parameter option for one of the four TUF roles. Each gene was limited to a range of 0 to 62 and the integer value was mapped to a specific SHBS parameter choice. A random initial choice was selected.

There are many optional parameters that can be set when using a genetic algorithm. The specific parameters that had a noticeable impact on the selection performance for this application are shown in Table 6.1. Changes in the algorithm configuration can have a significant impact on the time to converge and the tendency for the genetic search to get

Number of Genes:	4
Type of Genes:	integer
Maximum Iterations:	2000
Number of Parents Mating:	20
Solutions per Population:	40
Keep Elitism (best solutions retained):	5
Parent Selection Type:	rank
Keep Parents:	2
Crossover Type:	two_points
Mutation Type:	random
Mutation Genes:	2

Table 6.1: Genetic algorithm options used for SHBS parameter selection.

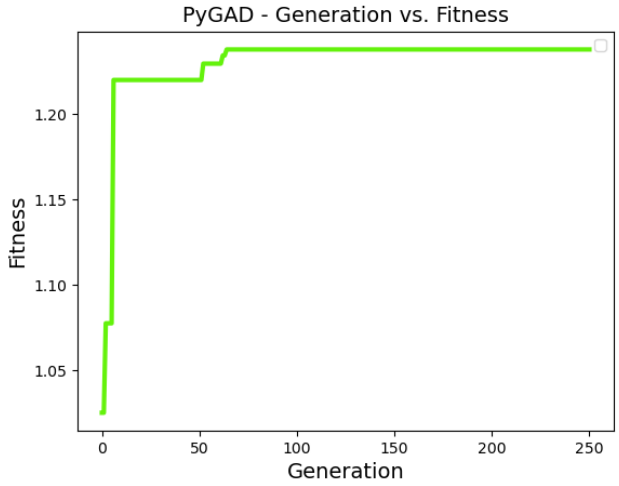

Figure 6.1: Optimization results using PyGAD.

stuck at a local minima.

Figure 6.1 is an example of the progress of the algorithm showing the value of the fitness function over 250 generations. It shows that the algorithm was able to select a local minimum parameter set that satisfied the optimization criteria. This approach may be valuable for any software update deployment where there are specific performance requirements. One advantage of using a genetic algorithm for optimization is that it is relatively easy to add additional optimization criteria that can guide the outcome.

Figure 6.2 shows optimization results for 1 Root role and for 5 Root roles using simula-

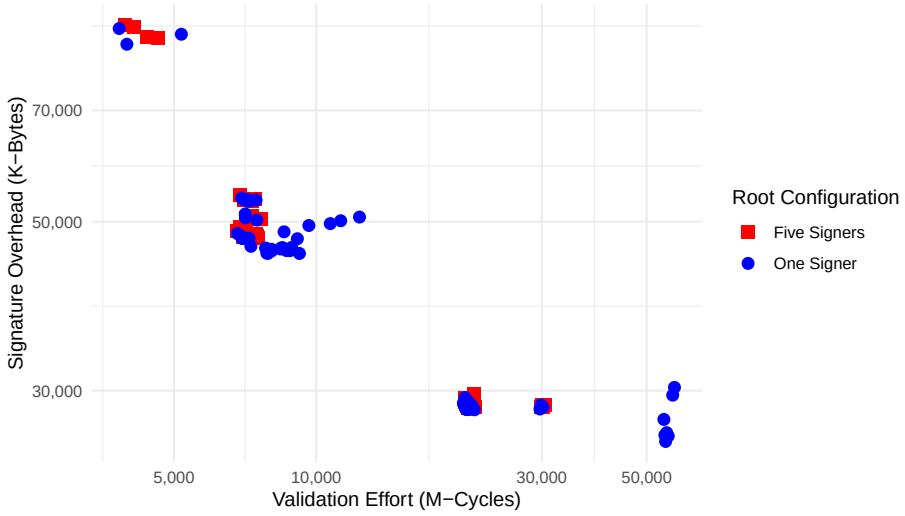

Figure 6.2: Optimization Results for 1 and 5 roots.

tion data for Device A. These points are plotted as a log-log scatter, similar to the results shown in Chapter 5. Each point on this graph represents a unique combination of different SHBS parameter sets assigned to each of the roles in TUF. It is valuable to observe in Figure 6.2 that there is a mix of points from both the 1 Root signer and the 5 root signer sets. This is similar to the finding in Chapter 5 where it was observed that the switch to five singers for the Root role had a limited impact on overall performance. This result shows that even when optimized to use unique SHBS parameters for each role, the relative impact of adopting multiple signatures for the TUF Root role is minimal.

Figure 6.3 compares the optimization results against the values obtained with a fixed algorithm selected for each role. The data points labeled as "Fixed" are identical to the points shown in Figure 5.7. These points represent simulation results for all approved SHBS parameter sets where all four TUF roles were assigned to the same parameters. The points labeled as "Optimal" represent simulations where the optimization method was used to select unique parameters for each TUF role. These results were obtained by running the optimization 50 times, each time with a randomly selected value for w_v in the range of 0 to 1. Assignment of this value forces the optimization algorithm to favor minimal cumulative download size, minimal cumulative validation effort, or a linear combination of the two.

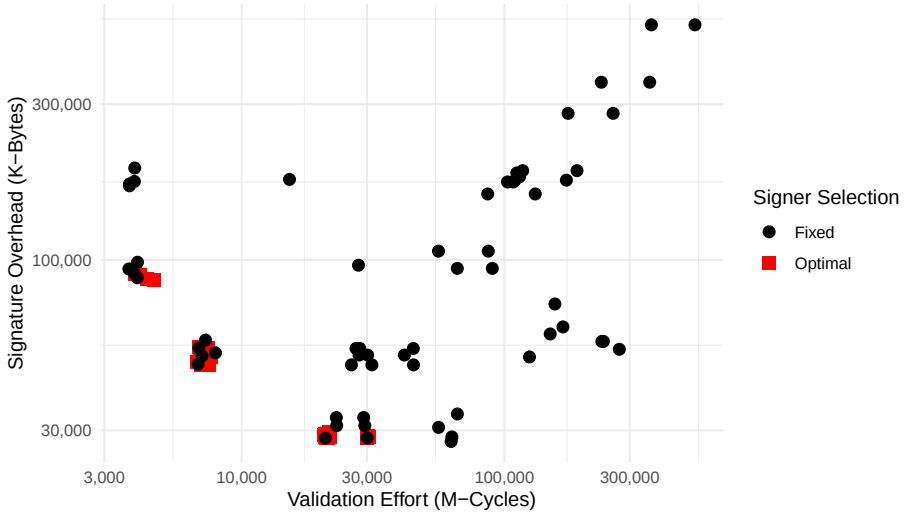

Figure 6.3: Optimization results compared with fixed algorithm results.

Notice that all of the "Optimal" points are positioned along the Pareto Frontier. This shows that the optimization method is effectively finding near-optimal solutions. It is also clear that the results from the optimization method are not noticeably better than the results that are possible when the same parameters are used for all roles. This suggests that the added complexity of assigning unique parameters has limited benefit.

6.4 Chapter Summary

Chapter 5 demonstrated that it is feasible to integrate SHBS with the TUF method for software updates. The simulations run to demonstrate feasibility enforced a simplifying assumption that all TUF roles would be implemented using the exact same SHBS parameters. In this chapter this simplifying assumption was removed to explore the 8.5×10^{37} possible combinations of parameter choices when there are four roles, each with 63 options. It has been shown that a commonly available implementation of a genetic algorithm can be applied as an effective way to make a choice from this complex array of parameter options. Comparison of optimized results with the exhaustive search method introduced in Chapter 5 indicates that the benefit in assigning unique parameters to each role is minimal. The results

do support that an optimization method can be used to automate the selection of parameters and that the result is no worse than the best choices found using an exhaustive search.

Chapter 7: Conclusions and Future Research Recommendations

This research has explored the combination of SHBS and TUF as a resilient post-quantum approach to software updates for CI systems. It has also advocated for the use of multiple signatures to validate software updates and provided a BCR model to evaluate the benefit of applying two or more signatures to an update package. An optimization model has also been proposed to help in choosing parameters for complex SHBS systems. The following section will review these results in the context of the original research questions and each corresponding hypotheses. Limitations of these results and opportunities for future work are also examined.

7.1 Conclusions

The challenge of defining a resilient solution for CI software updates was divided into three research questions, each with a corresponding hypothesis and methodology as outlined in Figure 3.1. Each of the proposed research methodologies has been applied to test the individual hypotheses. Each research question may be evaluated individually with the results then combined to reach a conclusion on the original problem statement.

The first hypothesis proposed the use of a statistical model to show the impact of using two or more signatures on the risk of a successful attack. Chapter 4 explored the use of a multinomial probability model to evaluate the relative risk of using multiple signers. The resulting model showed that when multiple signatures are used by a client system to evaluate a software update, the probability of accepting a malicious update can be significantly reduced. A BCR calculation was proposed to evaluate the relative benefit of adopting two or more signatures. This result addresses Research Question 1 which seeks a measure of relative probability for a successful attack when two or more signers are required to approve a software update package. This insight can inform design decisions for many CI systems

where the cost of an adverse event resulting from a client system accepting a malicious software update may be catastrophically high.

The second hypothesis suggested that simulation results would be able to measure the impact of integrating SHBS, a NIST approved and recommended post-quantum set of algorithms, with TUF, an update methodology designed to address observed threats against the software supply chain. Chapter 5 introduced a Python simulation model which was used to simulate this integrated solution. Simulation results were shown for two realistic CI systems, showing that the combined system requires approximately 2.6 times as much computational effort and 21 times the network traffic as a conventional implementation of TUF. These results address the second research question which seeks metrics for the combined system. It was further shown that multiple signers can be used to protect the Root role, the most critical of the four roles in TUF. Simulation results show that with five signers assigned to the Root role, the relative impact on the cumulative download size is small, but there is a larger impact on the cumulative validation effort.

The third hypothesis claims that a genetic algorithm can be used as an effective way to recommend a near-optimal set of parameters, accounting for both network utilization and system resources needed for validation. In Chapter 6, it was shown that PyGAD, a popular Python implementation of a genetic algorithm for optimization, can be configured to choose among the 8.5×10^{37} total choices of parameters that must be considered when assigning unique values to each of the four roles required for the TUF update method. Specific parameters are recommended for configuring the genetic optimization process with results showing that the system converges on an effective solution. Optimization results were shown for cases with both one and five Root role signers, once again that the impact of adopting five signers is minimal. Results also demonstrated that the genetic algorithm approach can find near-optimal solutions.

The combination of these results demonstrates that SHBS can be effectively combined with TUF to create a viable update solution for CI systems. This work supports the

recommendation in CNSA which encourages deployers of systems that will be in the field for a long period of time to adopt post-quantum algorithms now. SHBS are an attractive post-quantum option because they rely on a minimum set of few assumptions.

The specific combination of SHBS with TUF effectively mitigates many of the client-side challenges associated with stateful signatures. Specifically, it eliminates concerns over the limited number of signatures available for each key pair since the TUF model provides a secure method for distributing replacement keys. The proposed approach represents a practical and resilient post-quantum software update option for CI systems.

7.2 Limitations

The probability model developed in Chapter 4, and the corresponding BCR calculation method, depend on many parameters that are difficult to obtain or accurately estimate. A Bayesian update method is proposed to improve the probability estimates over time, but this may be of limited value for the design a new system with no historical threat information. Survey data with expert insights and detailed historical attack data from comparable could be used to extend this work to provide more specific risk and cost analysis which could be used to recommend a specific number of signers and quorum value.

The simulation model developed in Chapter 5 only applies to the TUF update method when combined with one of the approved SHBS algorithms. While there are sound arguments for limiting the selection to only the set of approved SHBS algorithms, alternatives such as SPHINCS or other approved post-quantum algorithms could also be evaluated through simulation. Similarly, the optimization method demonstrated in Chapter 6 is restricted to choosing among the approved parameter sets for SHBS algorithms. Alternative post-quantum algorithms are not considered.

Several of the successful attacks against software supply chains introduced in Chapter 2 were executed by injecting malicious code into the secure development process well ahead of digital signature application. An improved digital signature method will have limited

benefit against this type of attack. It is possible that with independent signers each evaluating the plausibility and integrity of each update package, they may discover malicious code that was inserted into an update, but the modeling work introduced in this research does not improve upon existing methods for performing detection.

7.3 Future Research

Section 4.11 proposes a method for applying Bayesian estimation to incorporate expert judgment with experimental results to estimate the actual probability of attack. Recall that Paté-Cornell et al. (2018) had performed an analysis of past attacks against CI systems. That work described the many challenges involved in estimating actual probability of a future attack. As an alternative the authors identified precursor evidence that may suggest a future attack may be more likely. Separate work by Bencsath et al. (2012) found similar predictive activity ahead of the "stuxnet" event. It may be possible to use the frequency of these predictive events to feed into the Bayesian estimation method proposed in Section 4.11. This approach may yield a predictor of future attacks that can be used to produce more accurate estimates of the risk exposure for some systems.

Section 2.9 proposed the idea of applying a BFT model for software update validation outside of a CI environment. In the case of an enterprise network where there are many systems that may receive an identical update package at the same time and where there are direct communication paths between systems, they could use a consensus mechanism to decide on the validity of the update package. This may take the form of a blockchain based model for validating the integrity and authenticity of a software update package.

The sequence of update events used in all of the simulations was based on just two representative device types. This is sufficient to show the relative cost of adopting SHBS and demonstrating that combining these algorithms with TUF can overcome the limited number of signatures. In future work it would be valuable to extend the simulation to include randomly timed unplanned and critical update events. A Mote Carlo simulation environment

would allow designers to evaluate parameter options in the presence of these unplanned events. This may result in changes to the selected parameter choice based on the impact of the additional required updates.

The collection of post-quantum algorithms considered for combination with TUF could be extended to produce a hybrid that can incorporate multiple algorithms. The would be a hybrid signature validation approach that relies on the strict adherence to a validation policy to ensure consistent results. A benefit of this approach is that it could incorporate traditional signature methods such as ECDSA along with post-quantum algorithms. The genetic algorithm optimization method could be extended to include this broader array of options to provide a near-optimal, risk-adjusted combination of algorithms that meet a demanding set of requirements for any mission-critical system that requires highly resilient long-term protection.

Bibliography

ISO/IEC/IEEE 12207:2017(E) First edition 2017-11: : ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes. International standard, ISO 12207:2017(E), IEC 12207:2017(E), 12207:2017(E). IEEE, New York, USA, 2017. ISBN 1-5044-4253-9.

H. Abdo, M. Kaouk, J.-M. Flaus, and F. Masse. A safety/security risk analysis approach of industrial control systems: A cyber bowtie – combining new version of attack tree with bowtie analysis. *Computers & Security*, 72:175–195, 2018. ISSN 0167-4048.

Diogo Avelas, Hasan Heydari, Eduardo Alchieri, Tobias Distler, and Alysson Bessani. Probabilistic byzantine fault tolerance. In *PODC 2024 - Proceedings of the 2024 ACM Symposium on Principles of Distributed Computing*, pages 170–181, New York, NY, USA, 2024. ACM. ISBN 9798400706684.

Jeffrey Avery and John Ross Wallrabenstein. Formally modeling deceptive patches using a game-based approach. *Computers & security*, 75:182–190, 2018. ISSN 0167-4048.

Sean S. Baggott and Joost R. Santos. A risk analysis framework for cyber security and critical infrastructure protection of the u.s. electric power grid. *Risk analysis*, 40(9): 1744–1761, 2020. ISSN 0272-4332.

Gustavo Banegas, Koen Zandberg, Emmanuel Baccelli, Adrian Herrmann, and Benjamin Smith. Quantum-resistant software update security on low-power networked embedded devices. volume 13269 LNCS, pages 872 – 891, Virtual, Online, 2022. URL `http://dx.doi.org/10.1007/978-3-031-09234-3_43`.

Yaser Baseri, Vikas Chouhan, and Abdelhakim Hafid. Navigating quantum security risks in networked environments: A comprehensive study of quantum-safe network protocols. *Computers & Security*, 142:103883–, 2024. ISSN 0167-4048.

Boldizsar Bencsath, Gabor Pek, Levente Buttyan, and Mark Felegyhazi. The cousins of stuxnet: Duqu, flame, and gauss. *Future internet*, 4(4):971–1003, 2012. ISSN 1999-5903.

Christian Berger, Signe Schwarz-Rüsch, Arne Vogel, Kai Bleeke, Leander Jehl, Hans P Reiser, and Rüdiger Kapitza. Sok: Scalability techniques for bft consensus. 2023.

Nina Bindel and Britta Hale. A note on hybrid signature schemes. Cryptology ePrint Archive, Paper 2023/423, 2023. URL `https://eprint.iacr.org/2023/423`. `https://eprint.iacr.org/2023/423`.

Ayat-Allah Bouramdane. Cyberattacks in smart grids: Challenges and solving the multi-criteria decision-making for cybersecurity options, including ones that incorporate artificial intelligence, using an analytical hierarchy process. *Journal of cybersecurity and privacy*, 3(4):662–705, 2023. ISSN 2624-800X.

Ioana Boureanu. Formally verifying the security and privacy of an adopted standard for software-update in cars: Verifying uptane 2.0. In *Conference proceedings - IEEE International Conference on Systems, Man, and Cybernetics*, pages 1301–1306, 2023. ISBN 9798350337020.

Hugh Boyes. Cybersecurity and cyber-resilient supply chains. *Technology innovation management review*, 5(4):28–34, 2015. ISSN 1927-0321.

Benedikt Brecht, Dean Therriault, Roger Motz, Virendra Kumar, Rob Lambert, Bill Lattin, Brian Romansky, and William Whyte. Elector-based root management system to manage a public key infrastructure, 03 2016.

Johannes. Buchmann and Jintai. Ding. *Post-quantum cryptography : second international workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008 : proceedings*. Lecture notes in computer science, 5299. Springer, Berlin, 1st ed. 2009. edition, 2008. ISBN 1-282-00091-8.

Denis Butin. Hash-based signatures: State of play. *IEEE Security & Privacy*, 15(4):37–43, 2017. ISSN 1540-7993.

Fabio Campos, Tim Kohlstadt, Steffen Reith, and Marc Stöttinger. Lms vs xmss: Comparison of stateful hash-based signature schemes on arm cortex-m4. In *Progress in Cryptology - AFRICACRYPT 2020: 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20 – 22, 2020, Proceedings*, page 258–277, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-51937-7. doi: 10.1007/978-3-030-51938-4_13. URL https://doi.org/10.1007/978-3-030-51938-4_13.

Daniel E Capano. Understand the cyber-attack lifecycle. *Control Engineering*, 66(7):32–33, 2019. ISSN 0010-8049.

Lidong Chen, David McGrew, and Chris Mitchell. State management for hash-based signatures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10074 of *Lecture Notes in Computer Science*, pages 244–260. Springer International Publishing AG, Switzerland, 2016. ISBN 9783319490991.

Ivan Cindrić, Marko Jurčević, and Tamara Hadjina. Mapping of industrial iot to iec 62443 standards. *Sensors (Basel, Switzerland)*, 25(3):728–, 2025. ISSN 1424-8220.

IEEE 1609 Committee. Standard for wireless access in vehicular environments (wave) - certificate management interfaces for end entities. Technical report, Institute of Electrical and Electronics Engineers, New York, NY, USA, 2022.

David Cooper, Andrew Regenscheid, and Murugiah Souppaya. Security considerations for code signing, 2018-01-26 00:01:00 2018. URL https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=925081.

David A. Cooper, Daniel C. Apon, Quynh H. Dang, Michael S. Davidson, Morris J. Dworkin, and Carl A. Miller. Sp 800-208: Recommendation for stateful hash-based signature schemes. Technical report, 10 2020.

Bart Coppens, Bjorn De Sutter, and Koen De Bosschere. Protecting your software updates. *IEEE Security & Privacy*, 11(2):47–54, 2013. ISSN 1540-7993.

Jintai Ding and Albrecht Petzoldt. Current state of multivariate cryptography. *IEEE Security & Privacy*, 15(4):28–36, 2017. ISSN 1540-7993.

Konstantin Fischesr, Ivana Trummova, Phillip Gajland, Yasemin Acar, Sascha Fahl, and M Angela Sasse. Adoption challenges for cryptographic protocols. 2024.

Stefan Lukas Gazdag, Markus Friedl, and Daniel Loebenberger. Post-quantum software updates. In *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, volume 294, pages 459–472, 2019. ISBN 3885796880.

Daniel E. Geer. Complexity is the enemy, 2008. ISSN 1540-7993.

Marcus Geiger, Jochen Bauer, Michael Masuch, and Jorg Franke. An analysis of black energy 3, crashoverride, and trisis, three malware approaches targeting operational technology systems. In *Proceedings (IEEE International Conference on Emerging Technologies and Factory Automation)*, volume 1, pages 1537–1543. IEEE, 2020. ISBN 172818956X.

M. Grabowski, J.R.W. Merrick, J.R. Harrold, T.A. Massuchi, and J.D. van Dorp. Risk modeling in distributed, large-scale systems. *IEEE transactions on systems, man and cybernetics. Part A, Systems and humans*, 30(6):651–660, 2000. ISSN 1083-4427.

Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou. *Computer Security - ESORICS 2010: 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, volume 6345 of *Lecture Notes in Computer Science*. Springer Nature, Netherlands, 2010 edition edition, 2010. ISBN 9783642154973.

Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. Investigating package related security threats in software registries. In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2023-, pages 1578–1595. IEEE, 2023. ISBN 1665493364.

Masayuki Hatta. The nebraska problem in open source software development. *Annals of Business Administrative Science*, 21(5):91–102, 2022. ISSN 1347-4464.

Jorge L. Hernandez-Ardieta, Ana I. Gonzalez-Tablas, Jose M. de Fuentes, and Benjamin Ramos. A taxonomy and survey of attacks on digital signatures. *Computers & Security*, 34:67–112, 2013. ISSN 0167-4048.

Russ Housley. Cryptographic Message Syntax (CMS). RFC 5652, September 2009. URL `https://www.rfc-editor.org/info/rfc5652`.

Andreas Huelsing, Denis Butin, Stefan Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, IETF, May 2018. URL https://www.rfc-editor.org/rfc/rfc8391.txt.

Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for xmss. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8128, pages 194–208, 2013. ISBN 9783642405877.

IEEE and The Open Group. *The Open Group Base Specifications Issue 7: IEEE Std 1003.1-2008 (POSIX.1)*. The IEEE and The Open Group, 2008. ISO/IEC 9945:2009, IEEE Std 1003.1-2008, The Open Group Technical Standard Base Specifications, Issue 7.

Donny Jackson. Lessons not learned from software supply-chain attacks. *Urgent Communications*, 2023. ISSN 1946-4384.

Simon Josefsson and Jim Schaad. IETF RFC 8554: Leighton-micali hash-based signatures. RFC 8554, IETF, April 2019. URL https://www.rfc-editor.org/rfc/rfc8554.txt.

Eric D. Knapp. *Industrial Network Security : Securing Critical Infrastructure Networks for Smart Grid, SCADA, and Other Industrial Control Systems*. Syngress, Cambridge, MA, third edition. edition, 2024. ISBN 9780443137389.

Neal Koblitz and Alfred Menezes. A riddle wrapped in an enigma. *IEEE Security & Privacy*, 14(6):34–42, 2016. ISSN 1540-7993.

Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, volume 2023-, pages 1509–1526. IEEE, 2023a. ISBN 1665493364.

Piergiorgio Ladisa, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez, and Olivier Barais. Journey to the center of software supply chain attacks. *IEEE Security & Privacy*, 21(6):2–17, 2023b. ISSN 1540-7993.

Thomas Lam and Nicholas Chaillan. Dod enterprise devsecops reference design: Version 1.0. *Defense AR Journal*, 28(2):242, 04 2021. URL http://proxygw.wrlc.org/login?url=https://www.proquest.com/scholarly-journals/dod-enterprise-devsecops-reference-design-version/docview/2542756188/se-2. Copyright - Copyright Superintendent of Documents Apr 2021; Last updated - 2023-11-06.

Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM transactions on programming languages and systems*, 4(3):382–401, 1982. ISSN 0164-0925.

Antonio Langiu, Carlo Alberto Boano, Markus Schuss, and Kay Romer. Upkit: An open-source, portable, and lightweight update framework for constrained iot devices. In *2019*

IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pages 2101–2112. IEEE, 2019. ISBN 9781728125190.

Gianluca Lax, Francesco Buccafurri, and Gianluca Caminiti. Digital document signing: Vulnerabilities and solutions. *Information security journal.*, 24(1-3):1–14, 2015. ISSN 1939-3555.

Michael Lesk. Trust, but verify. *IEEE Security & Privacy*, 12(6):94–96, 2014. ISSN 1540-7993.

E. Levy. Poisoning the software supply chain. *IEEE Security & Privacy*, 1(3):70–73, 2003. ISSN 1540-7993.

Fabio Massacci, Trent Jaeger, and Sean Peisert. Solarwinds and the challenges of patching: Can we ever stop dancing with the devil? *IEEE Security & Privacy*, 19(2):14–19, 2021. ISSN 1540-7993.

John T. J. Mathieson, Thomas Mazzuchi, and Shahram Sarkani. The systems engineering devops lemniscate and model-based system operations. *IEEE Systems Journal*, 15(3): 3980–3991, 2021. ISSN 1932-8184.

T.A. Mazzuchi and R. Soyer. A bayes method for assessing product-reliability during development testing. *IEEE transactions on reliability*, 42(3):503–510, 1993. ISSN 0018-9529.

Joseph Mitola and Mark Prys. Cyber oriented digital engineering. *Systems engineering*, 27 (1):109–119, 2024. ISSN 1098-1241.

Hadi Mohammadi. A systems engineering framework for implementing a security and critical patch management process in diverse environments (academic departments' workstations), 2014.

Dustin Moody and Angela Robinson. Cryptographic standards in the post-quantum era. *IEEE security & privacy*, 20(6):66–72, 2022. ISSN 1540-7993.

National Institute of Standards and Technology. Sp 800-53r5: Security and privacy controls for information systems and organizations. Technical report, 9 2020.

National Security Agency. Announcing the Commercial National Security Algorithm Suite 2.0. Technical Report U/OO/194427-22, National Security Agency, Fort Meade, MD, September 2022.

National Security Agency. The Commercial National Security Algorithm Suite 2.0 and quantum computing faq. Technical Report U/OO/194427-22, Ver. 2.0, National Security Agency, Fort Meade, MD, April 2024.

NIST. Guide for conducting risk assessments, 09 2012.

Arash Nourian and Stuart Madnick. A systems theoretic approach to the security threats in cyber physical systems applied to stuxnet. *IEEE transactions on dependable and secure computing*, 15(1):2–13, 2018. ISSN 1545-5971.

Arne Roar Nygård and Sokratis Katsikas. Sok: Combating threats in the digital supply chain. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396707. doi: 10.1145/3538969.3544421. URL https://doi.org/10.1145/3538969.3544421.

Chinenye Okafor, Taylor R Schorlemmer, Santiago Torres-Arias, and James C Davis. Sok: Analysis of software supply chain security by establishing secure design properties. 2024.

Mike Ounsworth, John Gray, Massimiliano Pala, and Jan Klaußner. Composite Signatures For Use In Internet PKI. Internet-Draft draft-ounsworth-pq-composite-sigs-10, Internet Engineering Task Force, October 2023. URL https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-sigs/10/. Work in Progress.

James Parker, Michael Hicks, Andrew Ruef, Michelle L. Mazurek, Dave Levin, Daniel Votipka, Piotr Mardziel, and Kelsey R. Fulton. Build it, break it, fix it: Contesting secure development. *ACM transactions on privacy and security*, 23(2):1–36, 2020. ISSN 2471-2566.

M.-Elisabeth Paté-Cornell, Marshall Kuypers, Matthew Smith, and Philip Keller. Cyber risk management for critical infrastructure: A risk analysis model and three case studies. *Risk analysis*, 38(2):226–241, 2018. ISSN 0272-4332.

Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. Perspectives on the solarwinds incident. *IEEE Security & Privacy*, 19(2):7–13, 2021. ISSN 1540-7993.

PKWARE, Inc. .zip file format specification: Appnote.txt. Technical specification, PKWARE, Inc., 2024. Version 6.3.10. Available online: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.

Fred L. Ramsey. A bayesian approach to bioassay. *Biometrics*, 28(3):841–858, 1972. ISSN 0006-341X.

Eleanor Rieffel and Wolfgang Polak. *Quantum computing : a gentle introduction*. Scientific and engineering computation. The MIT Press, Cambridge, Mass, 2011 - 2011. ISBN 9780262015066.

Ronald S. Ross, Victoria Pillitteri, Richard Graubart, Deborah Bodeau, and Rosalie McQuaid. Developing cyber-resilient systems: A systems security engineering approach, 2022-11-16 05:11:00 2022. URL https://csrc.nist.gov/pubs/sp/800/160/v2/r1/final.

Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on computer and communications security*, CCS '10, pages 61–72. ACM, 2010. ISBN 1450302459.

Bruce Schneier. Cryptography after the aliens land. *IEEE Security & Privacy*, 16(5):86–88, 2018. ISSN 1540-7993.

Jun Shao and Zhenfu Cao. A traceable threshold signature scheme with multiple signing policies. *Computers & Security*, 25(3):201–206, 2006. ISSN 0167-4048.

Murugiah Souppaya and Karen Scarfone. Sp 800-40r4: Guide to enterprise patch management planning: Preventive maintenance for technology. Technical report, 4 2022.

Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography (SAC) 2016*, volume 10532 of *LNCS*, pages 1–24. Springer, October 2017. URL `https://openquantumsafe.org`.

Chee-Wooi Ten, Govindarasu Manimaran, and Chen-Ching Liu. Cybersecurity for critical infrastructures: Attack and defense modeling. *IEEE transactions on systems, man and cybernetics. Part A, Systems and humans*, 40(4):853–865, 2010. ISSN 1083-4427.

Shiang-Feng Tzeng, Min-Shiang Hwang, and Cheng-Ying Yang. An improvement of nonrepudiable threshold proxy signature scheme with known signers. *Computers & Security*, 23(2):174–178, 2004. ISSN 0167-4048.

United States Department of Homeland Security. *NIPP 2013 : partnering for critical infrastructure security and resilience.* Homeland Security, Washington, D.C, 2013.

John Viega. Ten years of trustworthy computing: Lessons learned, 2011. ISSN 1540-7993.

Alexander Wagner, Felix Oberhansl, and Marc Schink. Extended version—to be, or not to be stateful: post-quantum secure boot using hash-based signatures. *Journal of cryptographic engineering*, 14(4):631–648, 2024. ISSN 2190-8508.

Carl Wallace, Sam Ashmore, and Russ Housley. Trust Anchor Management Protocol (TAMP). RFC 5934, August 2010. URL `https://www.rfc-editor.org/info/rfc5934`.

Gang Wang and Mark Nixon. Sok: Essentials of bft consensus for blockchains. In *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)*, pages 315–328. IEEE, 2023. ISBN 9798350339239.

Dominik Wermke, Jan H. Klemmer, Noah Wohler, Juliane Schmuser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. "always contribute back": A qualitative study on security challenges of the open source supply chain. In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2023-, pages 1545–1560. IEEE, 2023. ISBN 1665493364.

Samuel Stanley Wilks. *Mathematical statistics.* A Wiley publication in mathematical statistics. Wiley, New York, 1962.

Jiangshan Yu. Mitigating private key compromise, 2016.

Koen Zandberg, Kaspar Schleiser, Francisco Acosta, Hannes Tschofenig, and Emmanuel Baccelli. Secure firmware updates for constrained iot devices using open standards: A reality check. *IEEE access*, 7:71907–71920, 2019. ISSN 2169-3536.

Jiashuo Zhang, Jianbo Gao, Ke Wang, Zhenhao Wu, Yue Li, Zhi Guan, and Zhong Chen. Tbft: Efficient byzantine fault tolerance using trusted execution environment. In *IEEE International Conference on Communications (2003)*, pages 1004–1009. IEEE, 2022. ISBN 1538683474.

Qianru Zhu and Benjamin D. Leibowicz. A markov decision process approach for cost-benefit analysis of infrastructure resilience upgrades. *Risk analysis*, 42(7):1585–1602, 2022. ISSN 0272-4332.

Appendix A: XMSS and XMSS^{MT} Benchmark Data

Table A.1 contains benchmark data for the XMSS and XMSS^{MT} signature methods. This data was generated using the `libOQS` library Stebila and Mosca (2017). The "Algorithm ID" matches the algorithm identifiers defined in the `libOQS` code. Values for signature, public, and secret key lengths are reported by the library.

Validation time was calculated by executing the validation routine on an x86 system. The values shown are an approximation of the total cycles required for a single message valuation, shown as millions of cycles. The estimate was calculated by averaging the in-process time for 1,000 validation operations and converting the time to cycles using the system reported CPU frequency.

Algorithm ID	Number of Signatures $log_2(N)$	Signature Length (KB)	Public Key Length (B)	Secret Key Length (B)	Validation Time (M-Cycles)
XMSS-SHA2_10_256	10	2.44	68	1,377	4.62
XMSS-SHA2_16_256	16	2.63	68	2,097	4.28
XMSS-SHA2_20_256	20	2.75	68	2,577	4.63
XMSS-SHAKE_10_256	10	2.44	68	1,377	2.68
XMSS-SHAKE_16_256	16	2.63	68	2,097	2.88
XMSS-SHAKE_20_256	20	2.75	68	2,577	2.80
XMSS-SHA2_10_512	10	8.88	132	2,657	11.08
XMSS-SHA2_16_512	16	9.25	132	4,049	11.75
XMSS-SHA2_20_512	20	9.50	132	4,977	11.47
XMSS-SHAKE_10_512	10	8.88	132	2,657	10.52
XMSS-SHAKE_16_512	16	9.25	132	4,049	11.58
XMSS-SHAKE_20_512	20	9.50	132	4,977	11.81
XMSS-SHA2_10_192	10	1.46	52	1,057	3.08
XMSS-SHA2_16_192	16	1.60	52	1,609	3.03
XMSS-SHA2_20_192	20	1.69	52	1,977	3.00
XMSS-SHAKE256_10_192	10	1.46	52	1,057	2.14
XMSS-SHAKE256_16_192	16	1.60	52	1,609	2.37
XMSS-SHAKE256_20_192	20	1.69	52	1,977	2.36
XMSS-SHAKE256_10_256	10	2.44	68	1,377	3.21
XMSS-SHAKE256_16_256	16	2.63	68	2,097	3.11
XMSS-SHAKE256_20_256	20	2.75	68	2,577	2.90
XMSSMT-SHA2_20/2_256	20	4.85	68	6,002	9.26
XMSSMT-SHA2_20/4_256	20	9.03	68	10,942	17.72
XMSSMT-SHA2_40/2_256	40	5.47	68	9,604	8.94
XMSSMT-SHA2_40/4_256	40	9.66	68	15,256	19.45
XMSSMT-SHA2_40/8_256	40	18.04	68	24,520	36.77
XMSSMT-SHA2_60/3_256	60	8.20	68	16,633	13.49
XMSSMT-SHA2_60/6_256	60	14.48	68	24,511	26.69
XMSSMT-SHA2_60/12_256	60	27.04	68	38,099	54.67
XMSSMT-SHAKE_20/2_256	20	4.85	68	6,002	6.81
XMSSMT-SHAKE_20/4_256	20	9.03	68	10,942	11.30
XMSSMT-SHAKE_40/2_256	40	5.47	68	9,604	5.77
XMSSMT-SHAKE_40/4_256	40	9.66	68	15,256	12.08
XMSSMT-SHAKE_40/8_256	40	18.04	68	24,520	24.06
XMSSMT-SHAKE_60/3_256	60	8.20	68	16,633	8.89
XMSSMT-SHAKE_60/6_256	60	14.48	68	24,511	17.98
XMSSMT-SHAKE_60/12_256	60	27.04	68	38,099	37.33

Table A.1: XMSS and XMSSMT benchmark data

Appendix B: LMS and HSS Benchmark Data

Table B.1 contains benchmark data for the LMS and HSS signature methods. This data was generated using the `libOQS` library Stebila and Mosca (2017).

Validation time was calculated through executing the validation routine on an x86 system. The values shown are an approximation of the total cycles required for a single message validation, shown as millions of cycles. The estimate was calculated by averaging the in-process time for 1,000 validation operations and converting the time to cycles using the system reported CPU frequency.

Algorithm ID	Number of Signatures $log_2(N)$	Signature Length (KB)	Public Key Length (B)	Secret Key Length (B)	Validation Time (M-Cycles)
LMS_SHA256_H5_W1	5	8.48	60	64	0.35
LMS_SHA256_H5_W2	5	4.36	60	64	0.36
LMS_SHA256_H5_W4	5	2.30	60	64	0.71
LMS_SHA256_H5_W8	5	1.27	60	64	5.60
LMS_SHA256_H10_W1	10	8.64	60	64	0.38
LMS_SHA256_H10_W2	10	4.52	60	64	0.41
LMS_SHA256_H10_W4	10	2.45	60	64	0.70
LMS_SHA256_H10_W8	10	1.42	60	64	6.43
LMS_SHA256_H15_W1	15	8.80	60	64	0.39
LMS_SHA256_H15_W2	15	4.67	60	64	0.40
LMS_SHA256_H15_W4	15	2.61	60	64	0.73
LMS_SHA256_H15_W8	15	1.58	60	64	5.78
LMS_SHA256_H20_W1	20	8.95	60	64	0.40
LMS_SHA256_H20_W2	20	4.83	60	64	0.38
LMS_SHA256_H20_W4	20	2.77	60	64	0.71
LMS_SHA256_H20_W8	20	1.73	60	64	6.82
LMS_SHA256_H25_W4	25	2.92	60	64	0.75
LMS_SHA256_H5_W8_H5_W8	10	2.58	60	64	12.77
LMS_SHA256_H10_W4_H5_W8	15	3.77	60	64	16.02
LMS_SHA256_H10_W8_H5_W8	15	2.74	60	64	28.17
LMS_SHA256_H10_W2_H10_W2	20	9.08	60	64	1.57
LMS_SHA256_H10_W4_H10_W4	20	4.96	60	64	2.86
LMS_SHA256_H10_W8_H10_W8	20	2.89	60	64	24.51
LMS_SHA256_H15_W8_H5_W8	20	2.89	60	64	24.18
LMS_SHA256_H15_W8_H10_W8	25	3.05	60	64	15.37
LMS_SHA256_H15_W8_H15_W8	30	3.21	60	64	17.20
LMS_SHA256_H20_W8_H5_W8	25	3.05	60	64	15.81
LMS_SHA256_H20_W8_H10_W8	30	3.21	60	64	19.08
LMS_SHA256_H20_W8_H15_W8	35	3.36	60	64	24.19
LMS_SHA256_H20_W8_H20_W8	40	3.52	60	64	11.53

Table B.1: LMS and HSS benchmark data

Appendix C: ECDSA Benchmark Data

Table C.1 contains benchmark data for the ECDSA signature method.

Algorithm ID	Signature Length (KB)	Public Key Length (B)	Secret Key Length (B)	Validation Time (M-Cycles)
ECDSA-P256	0.06	65	32	0.23

Table C.1: ECDSA benchmark data

Appendix D: TUF Simulation Results

The following tables contain simulation results using the full suite of SHBS parameter sets for Device A and Device B. Each simulation executes all of the updates in the sequence defined for the corresponding device. Values for signatures and public keys represent the cumulative total of all content that is made available for client systems to download. Validation time is the compute effort (in millions of cycles) for one client to validate all signatures. Number of signatures is the cumulative number of signatures generated during the update sequence over the simulated lifespan of the device. These simulation results in each row represent the result where a single SHBS parameter set is used for all TUF roles.

Parameter Set	Total for Signatures (KB)	Total for Public Keys (KB)	Combined Signatures + Public Keys (KB)	Validation Time (M-Cycles)	Number of Signatures
XMSS-SHA2_10_256	23,730.5	2.7	23,733.1	44,898	9,720
XMSS-SHA2_16_256	25,529.3	0.3	25,529.6	41,601	9,711
XMSS-SHA2_20_256	26,743.2	0.3	26,743.4	44,989	9,711
XMSS-SHAKE_10_256	23,730.5	2.7	23,733.1	26,055	9,720
XMSS-SHAKE_16_256	25,529.3	0.3	25,529.6	27,965	9,711
XMSS-SHAKE_20_256	26,743.2	0.3	26,743.4	27,227	9,711
XMSS-SHA2_10_512	86,303.0	5.2	86,308.1	107,707	9,720
XMSS-SHA2_16_512	89,864.7	0.5	89,865.2	114,099	9,711
XMSS-SHA2_20_512	92,292.4	0.5	92,292.9	111,395	9,711
XMSS-SHAKE_10_512	86,303.0	5.2	86,308.1	102,299	9,720
XMSS-SHAKE_16_512	89,864.7	0.5	89,865.2	112,472	9,711
XMSS-SHAKE_20_512	92,292.4	0.5	92,292.9	114,685	9,711
XMSS-SHA2_10_192	14,162.3	2.0	14,164.4	29,916	9,720
XMSS-SHA2_16_192	15,514.8	0.2	15,515.0	29,415	9,711
XMSS-SHA2_20_192	16,425.2	0.2	16,425.4	29,121	9,711
XMSS-SHAKE256_10_192	14,162.3	2.0	14,164.4	20,753	9,720
XMSS-SHAKE256_16_192	15,514.8	0.2	15,515.0	22,990	9,711
XMSS-SHAKE256_20_192	16,425.2	0.2	16,425.4	22,917	9,711
XMSS-SHAKE256_10_256	23,730.5	2.7	23,733.1	31,182	9,720
XMSS-SHAKE256_16_256	25,529.3	0.3	25,529.6	30,153	9,711
XMSS-SHAKE256_20_256	26,743.2	0.3	26,743.4	28,135	9,711
XMSSMT-SHA2_20/2_256	47,066.1	0.3	47,066.4	89,915	9,711
XMSSMT-SHA2_20/4_256	87,730.9	0.3	87,731.2	172,038	9,711
XMSSMT-SHA2_40/2_256	53,154.4	0.3	53,154.7	86,783	9,711
XMSSMT-SHA2_40/4_256	93,819.3	0.3	93,819.5	188,852	9,711
XMSSMT-SHA2_40/8_256	175,148.9	0.3	175,149.2	357,062	9,711
XMSSMT-SHA2_60/3_256	79,584.7	0.3	79,584.9	130,972	9,711
XMSSMT-SHA2_60/6_256	140,581.9	0.3	140,582.2	259,143	9,711
XMSSMT-SHA2_60/12_256	262,576.3	0.3	262,576.6	530,878	9,711
XMSSMT-SHAKE_20/2_256	47,066.1	0.3	47,066.4	66,161	9,711
XMSSMT-SHAKE_20/4_256	87,730.9	0.3	87,731.2	109,768	9,711
XMSSMT-SHAKE_40/2_256	53,154.4	0.3	53,154.7	56,079	9,711
XMSSMT-SHAKE_40/4_256	93,819.3	0.3	93,819.5	117,306	9,711
XMSSMT-SHAKE_40/8_256	175,148.9	0.3	175,149.2	233,670	9,711
XMSSMT-SHAKE_60/3_256	79,584.7	0.3	79,584.9	86,322	9,711
XMSSMT-SHAKE_60/6_256	140,581.9	0.3	140,582.2	174,644	9,711
XMSSMT-SHAKE_60/12_256	262,576.3	0.3	262,576.6	362,485	9,711

Table D.1: TUF simulation results for Device A (part 1 of 2)

Parameter Set	Total for Signatures (KB)	Total for Public Keys (KB)	Combined Signatures + Public Keys (KB)	Validation Time (M-Cycles)	Number of Signatures
LMS_SHA256_H5_W1	85,021.9	72.9	85,094.8	3,486	10,021
LMS_SHA256_H5_W2	43,685.3	72.9	43,758.2	3,571	10,021
LMS_SHA256_H5_W4	23,017.0	72.9	23,089.9	7,071	10,021
LMS_SHA256_H5_W8	12,682.8	72.9	12,755.7	56,137	10,021
LMS_SHA256_H10_W1	83,986.9	2.3	83,989.2	3,726	9,720
LMS_SHA256_H10_W2	43,891.9	2.3	43,894.2	3,989	9,720
LMS_SHA256_H10_W4	23,844.4	2.3	23,846.7	6,799	9,720
LMS_SHA256_H10_W8	13,820.6	2.3	13,823.0	62,457	9,720
LMS_SHA256_H15_W1	85,426.5	0.2	85,426.7	3,742	9,711
LMS_SHA256_H15_W2	45,368.6	0.2	45,368.8	3,908	9,711
LMS_SHA256_H15_W4	25,339.6	0.2	25,339.9	7,066	9,711
LMS_SHA256_H15_W8	15,325.2	0.2	15,325.4	56,141	9,711
LMS_SHA256_H20_W1	86,943.8	0.2	86,944.0	3,904	9,711
LMS_SHA256_H20_W2	46,885.9	0.2	46,886.2	3,721	9,711
LMS_SHA256_H20_W4	26,857.0	0.2	26,857.2	6,875	9,711
LMS_SHA256_H20_W8	16,842.5	0.2	16,842.8	66,194	9,711
LMS_SHA256_H25_W4	28,374.3	0.2	28,374.6	7,283	9,711
LMS_SHA256_H5_W8_H5_W8	25,097.3	2.3	25,099.7	124,081	9,720
LMS_SHA256_H10_W4_H5_W8	36,605.9	0.2	36,606.2	155,613	9,711
LMS_SHA256_H10_W8_H5_W8	26,591.4	0.2	26,591.7	273,591	9,711
LMS_SHA256_H10_W2_H10_W2	88,195.6	0.2	88,195.8	15,200	9,711
LMS_SHA256_H10_W4_H10_W4	48,137.7	0.2	48,138.0	27,822	9,711
LMS_SHA256_H10_W8_H10_W8	28,108.8	0.2	28,109.0	237,977	9,711
LMS_SHA256_H15_W8_H5_W8	28,108.8	0.2	28,109.0	234,805	9,711
LMS_SHA256_H15_W8_H10_W8	29,626.1	0.2	29,626.4	149,303	9,711
LMS_SHA256_H15_W8_H15_W8	31,143.5	0.2	31,143.7	166,985	9,711
ECDSA-P256	606.9	0.1	607.1	1,341	9,711

Table D.2: TUF simulation results for Device A (part 2 of 2)

Parameter Set	Total for Signatures (KB)	Total for Public Keys (KB)	Combined Signatures + Public Keys (KB)	Validation Time (M-Cycles)	Number of Signatures
XMSS-SHA2_10_256	25,439.5	2.7	25,442.1	48,132	10,420
XMSS-SHA2_16_256	27,369.5	0.3	27,369.8	44,600	10,411
XMSS-SHA2_20_256	28,670.9	0.3	28,671.2	48,232	10,411
XMSS-SHAKE_10_256	25,439.5	2.7	25,442.1	27,931	10,420
XMSS-SHAKE_16_256	27,369.5	0.3	27,369.8	29,981	10,411
XMSS-SHAKE_20_256	28,670.9	0.3	28,671.2	29,190	10,411
XMSS-SHA2_10_512	92,518.2	5.2	92,523.4	115,464	10,420
XMSS-SHA2_16_512	96,342.4	0.5	96,342.9	122,323	10,411
XMSS-SHA2_20_512	98,945.2	0.5	98,945.7	119,425	10,411
XMSS-SHAKE_10_512	92,518.2	5.2	92,523.4	109,667	10,420
XMSS-SHAKE_16_512	96,342.4	0.5	96,342.9	120,579	10,411
XMSS-SHAKE_20_512	98,945.2	0.5	98,945.7	122,952	10,411
XMSS-SHA2_10_192	15,182.3	2.0	15,184.3	32,071	10,420
XMSS-SHA2_16_192	16,633.2	0.2	16,633.4	31,536	10,411
XMSS-SHA2_20_192	17,609.2	0.2	17,609.4	31,221	10,411
XMSS-SHAKE256_10_192	15,182.3	2.0	15,184.3	22,248	10,420
XMSS-SHAKE256_16_192	16,633.2	0.2	16,633.4	24,647	10,411
XMSS-SHAKE256_20_192	17,609.2	0.2	17,609.4	24,569	10,411
XMSS-SHAKE256_10_256	25,439.5	2.7	25,442.1	33,428	10,420
XMSS-SHAKE256_16_256	27,369.5	0.3	27,369.8	32,326	10,411
XMSS-SHAKE256_20_256	28,670.9	0.3	28,671.2	30,163	10,411
XMSSMT-SHA2_20/2_256	50,458.8	0.3	50,459.0	96,396	10,411
XMSSMT-SHA2_20/4_256	94,054.8	0.3	94,055.1	184,439	10,411
XMSSMT-SHA2_40/2_256	56,986.0	0.3	56,986.3	93,039	10,411
XMSSMT-SHA2_40/4_256	100,582.1	0.3	100,582.3	202,465	10,411
XMSSMT-SHA2_40/8_256	187,774.2	0.3	187,774.4	382,800	10,411
XMSSMT-SHA2_60/3_256	85,321.4	0.3	85,321.7	140,413	10,411
XMSSMT-SHA2_60/6_256	150,715.5	0.3	150,715.8	277,823	10,411
XMSSMT-SHA2_60/12_256	281,503.7	0.3	281,503.9	569,145	10,411
XMSSMT-SHAKE_20/2_256	50,458.8	0.3	50,459.0	70,930	10,411
XMSSMT-SHAKE_20/4_256	94,054.8	0.3	94,055.1	117,681	10,411
XMSSMT-SHAKE_40/2_256	56,986.0	0.3	56,986.3	60,121	10,411
XMSSMT-SHAKE_40/4_256	100,582.1	0.3	100,582.3	125,762	10,411
XMSSMT-SHAKE_40/8_256	187,774.2	0.3	187,774.4	250,514	10,411
XMSSMT-SHAKE_60/3_256	85,321.4	0.3	85,321.7	92,545	10,411
XMSSMT-SHAKE_60/6_256	150,715.5	0.3	150,715.8	187,233	10,411
XMSSMT-SHAKE_60/12_256	281,503.7	0.3	281,503.9	388,615	10,411

Table D.3: TUF simulation results for Device B (part 1 of 2)

Parameter Set	Total for Signatures (KB)	Total for Public Keys (KB)	Combined Signatures + Public Keys (KB)	Validation Time (M-Cycles)	Number of Signatures
LMS_SHA256_H5_W1	91,147.6	78.0	91,225.7	3,737	10,743
LMS_SHA256_H5_W2	46,832.8	78.0	46,910.8	3,828	10,743
LMS_SHA256_H5_W4	24,675.3	78.0	24,753.4	7,581	10,743
LMS_SHA256_H5_W8	13,596.6	78.0	13,674.7	60,181	10,743
LMS_SHA256_H10_W1	90,035.3	2.3	90,037.7	3,994	10,420
LMS_SHA256_H10_W2	47,052.8	2.3	47,055.2	4,276	10,420
LMS_SHA256_H10_W4	25,561.6	2.3	25,563.9	7,289	10,420
LMS_SHA256_H10_W8	14,815.9	2.3	14,818.3	66,955	10,420
LMS_SHA256_H15_W1	91,584.3	0.2	91,584.5	4,012	10,411
LMS_SHA256_H15_W2	48,638.9	0.2	48,639.1	4,190	10,411
LMS_SHA256_H15_W4	27,166.2	0.2	27,166.4	7,575	10,411
LMS_SHA256_H15_W8	16,429.9	0.2	16,430.1	60,187	10,411
LMS_SHA256_H20_W1	93,211.0	0.2	93,211.2	4,185	10,411
LMS_SHA256_H20_W2	50,265.6	0.2	50,265.8	3,989	10,411
LMS_SHA256_H20_W4	28,792.9	0.2	28,793.2	7,370	10,411
LMS_SHA256_H20_W8	18,056.6	0.2	18,056.8	70,965	10,411
LMS_SHA256_H25_W4	30,419.6	0.2	30,419.9	7,808	10,411
LMS_SHA256_H5_W8_H5_W8	26,904.8	2.3	26,907.1	133,017	10,420
LMS_SHA256_H10_W4_H5_W8	39,244.6	0.2	39,244.8	166,830	10,411
LMS_SHA256_H10_W8_H5_W8	28,508.2	0.2	28,508.5	293,313	10,411
LMS_SHA256_H10_W2_H10_W2	94,553.0	0.2	94,553.3	16,296	10,411
LMS_SHA256_H10_W4_H10_W4	51,607.7	0.2	51,607.9	29,827	10,411
LMS_SHA256_H10_W8_H10_W8	30,135.0	0.2	30,135.2	255,131	10,411
LMS_SHA256_H15_W8_H5_W8	30,135.0	0.2	30,135.2	251,731	10,411
LMS_SHA256_H15_W8_H10_W8	31,761.7	0.2	31,761.9	160,066	10,411
LMS_SHA256_H15_W8_H15_W8	33,388.4	0.2	33,388.6	179,022	10,411
ECDSA-P256	650.7	0.1	650.8	1,438	10,411

Table D.4: TUF simulation results for Device B (part 2 of 2)

Appendix E: Python Source Code

Source code is included for multinomial calculations and for the primary classes and functions used to produce the simulation and optimization results. Some sections of the code have been modified to configure execution for specific scenarios. Comments indicate code segments that require modification for specific results.

E.1 Multinomial Calculation

The `combinations.py` script was used to generate multinomial probability calculations for many variations of digital signing options. This code can be modified to evaluate any number of signers. It automatically produces results for all quorum values.

```
from scipy.stats import multinomial
import math
import numpy

def ListCombinations(N):
#####################################################################
# Generates an array with sets of 3 positive integers that sum to N
#  Args:
#    N: The target sum
#  Returns:
#    A list of lists, where each inner list represents a
#    combination of 3 integers that add up to N.
  combinations = []
  for a in range(0, N+1):
    for b in range(0, N-a+1):
      c = N - a - b
      if c >= 0 and (a+b+c) == N:
        combinations.append([a, b, c])
  return combinations

def ProbOccurance(Pl, combinations):
    prob_list = []
    rv = multinomial(sum(combinations[0]), Pl)
    for c in combinations:
        prob_list.append([c, rv.pmf(c)])
    return prob_list

def Strategy(Q, combinations):
    strat_list = []
```

```
    for c in combinations:
        # If the malicious signers have quorum, then you are
        # malicious
        if c[0][1] >= Q:
          s = 1
        # Else, if the valid signers have quoram, then you are
        # valid. Note that when they don't have quoram, malicious
        # signers act truthful
        elif (c[0][0]+c[0][1]) >= Q:
          s = 0
        # Otherwise, you are undecided (no quoram)
        else:
          s = 2
        strat_list.append([c, s])
    return strat_list

def Impact(combinations):
    pi = [0, 0, 0] # 0-T, 1-C, 2-D
    for c in combinations:
        pi[c[1]] = pi[c[1]] + c[0][1]
    return pi

def main():
  N_max = 51

  Pm = numpy.float128(0.1)
  Px = numpy.float128(0.2)

  Pt = 1 - Pm - Px
  Plist = [Pt, Pm, Px]

  print("[Pt, Pm, Px] = ", Plist);

  for n in range(1, N_max+1):
    list_of_sums = ListCombinations(n)
    prob_list = ProbOccurance(Plist, list_of_sums)
    for q in range(1, n+1):
      strat_list = Strategy(q, prob_list)
      prob_impact = Impact(strat_list)
      p_error = numpy.float128(1-prob_impact[0])
      log_p = numpy.float128(0)
      log_p = (-1) * numpy.log10(p_error)
      print("(", n, ",", q, ",", log_p,
            -numpy.log10(prob_impact[1]),
            -numpy.log10(prob_impact[2]), ")")

if __name__ == "__main__":
  main()
```

E.2 Support Packages

The package `TUFroles.py` defines a `TUFRepository` class that can be used to build a virtual repository and simulate a sequence of software update events. The `TUFrole` class is a generalization of a role that can have a named algorithm with properties such as the maximum number of signatures available. These classes keep track of the state of each role during a simulation and account for the total number of signatures and the cumulative impact on the client system.

```
# TUF_roles.py
#
# Define the roles in TUF.

import csv
from decimal import Decimal

# Algorithm Class
class SignatureAlgorithm:
    def __init__(self, name, sig_size, pk_size, max_sigs, cost):
        self.name = name
        self.sig_size = sig_size
        self.pk_size = pk_size
        self.max_sigs = max_sigs
        self.cost = cost

    def __str__(self):
        return(self.name)

    # Reads a list of algorithms from a CSV file.  Assumes that the
    # file has header rows that exactly match the filed names
    # listed here.  Returns the list of algorithms that were listed
    # in the CSV file.
    @staticmethod
    def LoadFromCSV(file):
        algorithms = []
        with open(file, 'r') as csvfile:

            # This section will read-ahead the first 1024 bytes of
            # the CSV to capture the field names, then clean up the
            # field names before re-loading the file and using
            # those field names.
            sniffer = csv.Sniffer()
            dialect = sniffer.sniff(csvfile.read(1024))
            csvfile.seek(0)
            reader = csv.DictReader(csvfile, dialect=dialect)
            fieldnames = [name.strip() for name in reader.fieldnames]
            reader = csv.DictReader(csvfile, fieldnames=fieldnames)
```

```
            #reader = csv.DictReader(csvfile)
            for row in reader:
                new_alg = \
                SignatureAlgorithm(row['Name'],
                    int(row['Signature Size']),
                    int(row['Public Key Size']),
                    int(Decimal(row['Max Signatures'])),
                    float(row['Computational Cost']))
                algorithms.append(new_alg)
        return(algorithms)

    # Finds a named algorithm in a list, returns the requested
    # algorithm (if it is present in the list)
    @staticmethod
    def FindInList(alg_name, algorithms):
        for alg in algorithms:
            if alg.name == alg_name:
                return(alg)
        raise ValueError("Requested algorithm type not found.")


# TUF Role
class TUFrole:
    def __init__(self, name, role_type, algorithm):
        self.name = name
        # identifier for the role
        self.role_type = role_type
        # role type (Root, Timestamp, etc)
        self.algorithm = algorithm
        # algorithm used for signatures
        self.num_sigs = 0
        # initialize signature count
        self.reserve = False
        # indicate that this role is not in reserve
        self.pending = True
        # indicate that role requires an update
        self.rollover = True
        # idicate rollover is required

    def __str__(self):
        return(f"{self.role_type} : {self.name} :
          sigs = {self.num_sigs}")

    def SignFile(self):
        self.num_sigs = self.num_sigs + 1

    def setReserve(self, state):
        self.reserve = state

    def getReserve(self):
        return(self.reserve)

# Repository
```

```
class TUFRepository:
    def __init__(self, name):
        self.name = name
        self.TUFroles = []
        self.accum_sig_size = 0
        self.accum_pk_size = 0
        self.accum_cost = 0.0
        self.accum_signatures = 0
        self.update_root = True
           # indicates that a new root is needed

    def __str__(self):
        return(self.name)

    def showStats(self):
        print(f"Repo \'{self.name}\' has accumulated:")
        print(f"     {self.accum_sig_size} bytes of signatures")
        print(f"     {self.accum_pk_size} bytes of public keys")
        print(f"     {self.accum_cost} units of cost")
        print(f"     {self.accum_signatures} total signatures")

def showStatsCSV(self):
        # <repo name>, <algorithm>, <sig bytes>,
        # <pk bytes>, <net cost>, <total sigs>
        print("%s, %d, %d, %d, %f, %d" %
              (self.name,
               self.accum_sig_size,
               self.accum_pk_size,
               (self.accum_sig_size + self.accum_pk_size),
               self.accum_cost,
               self.accum_signatures))

    def getAccumulatedBytes(self):
        return (self.accum_sig_size + self.accum_pk_size)

    def getAccumulatedCost(self):
        return (self.accum_cost)

    def addRole(self, name, role_type, alg):
        if (role_type == "Root" or
            role_type == "Timestamp" or
            role_type == "Snapshot" or
            role_type == "Target"):
            self.TUFroles.append(TUFrole(name, role_type, alg))
            self.update_root = True
              # indicates that a root update is needed
            for r in self.TUFroles:
                if r.name == name and r.role_type == role_type:
                    r.rollover = True
                    r.pending = True

    def listRoles(self):
        print(f"Repo \'{self.name}\' roles:")
        for r in self.TUFroles:
```

```
        print(f"   {r}")

def removeRole(self, name):
    rval = 0
    for r in self.TUFroles:
        if r.name == name:
            self.TUFroles.remove(r)
            rval = rval + 1
    if rval > 0:
        self.update_root = True   # root update is needed
    return(rval)

# Check to see if any roles have run out of signatures
# If they have, then stage them for a roll-over on the next
# timestamp.
def doRollover(self):
    n_rollovers = 0
    for r in self.TUFroles:

        # if a rule has reached max signatures and it is
        # required to perform a signature onnext timestamp,
        # then stage it for a rollover
        if (r.rollover == True or
            (r.num_sigs == r.algorithm.max_sigs
            and r.pending == True)):
            r.rollover = True
            r.num_sigs = 0
            n_rollovers = n_rollovers + 1
    return(n_rollovers)

# Produce a timestamp.
# Note that ALL roles that are pending a signature operation
# or rollover are performed at this time.
def doTimestamp(self):

    # Perform a rollover - i.e. flag roles need to be replaced
    if self.doRollover() > 0 or self.update_root:
        # If a rollover is required, account for signatures +
        # account for public keys in the new root.txt file
        for r in self.TUFroles:
            self.accum_pk_size = self.accum_pk_size +
                                 r.algorithm.pk_size
            if r.role_type == "Root":
                self.accum_sig_size = (self.accum_sig_size +
                                       r.algorithm.sig_size)
                self.accum_cost = self.accum_cost +
                    r.algorithm.cost
                self.accum_signatures =
                    self.accum_signatures + 1
                r.num_sigs = r.num_sigs + 1
            r.rollover = False  # clear rollover flag
            self.update_root = False # clear an update flag

    # Update the Targets.txt files for any targets
```

```
    n_updates = 0
    for r in self.TUFroles:
        if (r.role_type == "Target" and
            r.pending == True and
            r.reserve == False):
            # accumulate target signatures
            self.accum_sig_size = self.accum_sig_size +
              r.algorithm.sig_size
            self.accum_cost = self.accum_cost +
              r.algorithm.cost
            self.accum_signatures =
              self.accum_signatures + 1
            r.num_sigs = r.num_sigs + 1
            r.pending = False
            n_updates = n_updates + 1

    # If any targets were updated, then also update Snaphot.txt
    if n_updates > 0:
        for r in self.TUFroles:
            if (r.role_type == "Snapshot" and
                r.reserve == False):
                self.accum_sig_size = (self.accum_sig_size +
                                       r.algorithm.sig_size)
                self.accum_cost = (self.accum_cost +
                                   r.algorithm.cost)
                self.accum_signatures =
                  self.accum_signatures + 1
                r.num_sigs = r.num_sigs + 1

    # For every timestamp role not flagged as a reserve role,
    # stage it for signing.
    n_stamps = 0
    for r in self.TUFroles:
        if r.role_type == "Timestamp" and r.reserve == False:
            self.accum_sig_size = (self.accum_sig_size +
                                   r.algorithm.sig_size)
            self.accum_cost = (self.accum_cost +
                               r.algorithm.cost)
            self.accum_signatures =
              self.accum_signatures + 1
            r.num_sigs = r.num_sigs + 1
            n_stamps = n_stamps + 1

# Stage an update for the target named "target"
# Returns a count of the number of matching targets found
def doUpdate(self, target):
    rval = 0
    #print(f"Doing an update on target \'{target}\'")
    # Search through all of the roles
    for r in self.TUFroles:
        # If a role name matches the target
        if r.name == target and r.role_type == "Target":
            #print(f"Activating update on target \'{target}\'")
            r.pending = True
```

```
                rval = rval + 1

        # If you found a matching target, also activate Snapshots
        if rval > 0:
            for r in self.TUFroles:
                if r.role_type == "Snapshot":
                    r.pending = True

        return(rval)
```

The TUFsequence class provides helper functions that load a list of events from a CSV file. The event list includes a set of date-time delimited actions that must be applied to a simulated repository, such as an update event. Events can also be included to force replacement of a key due to compromise.

```
# TUFsequence.py
# Class for managing a sequence of updates

import csv
from datetime import date, timedelta

class EventList:
    def __init__(self, start, end, csv_file):
        self.start_date = date.fromisoformat(start)
        self.end_date = date.fromisoformat(end)
        self.event_list = EventList.LoadEventCSV(csv_file)

    def GetDates(self):
        date_list = []
        for d in range(int((self.end_date -
                            self.start_date).days) + 1):
            date_list.append(self.start_date + timedelta(d))
        return (date_list)

    def GetEvents(self):
        return(self.event_list)

    @staticmethod
    def LoadEventCSV(event_file):
        events = []
        with open(event_file, 'r') as csvfile:
            reader = csv.reader(csvfile)
            for row in reader:
                # Assume that the first column is the list of
                # event dates.  Also assumes that the dates are
                # in ISO format.
                events.append(date.fromisoformat(row[0]))
        return(events)
```

E.3 Main Simulator

The `TUFsim.py` file is the main function that is used to drive the simulator. This can be configured to handle any configuration of TUF roles in a single simulation. The version shown here builds a TUF repository that has five root signers and a single instance of all other roles. This code relies on CSV files to provide a list of algorithm parameters and an update sequence for each device type.

```
# TUF_sim.py
# Basic simulator, uses the TUF simulation classes

from datetime import datetime, timedelta

# Function to create a sequence of days between two defined dates
def date_range(start_date, end_date):
    for n in range(int((end_date - start_date).days) + 1):
        yield start_date + timedelta(n)


from TUFroles import *
from TUFsequence import *

def run_sequence(alg1, verbose, start, end, seq_file, dev_name):
    r = TUFRepository(dev_name)

    r.addRole("Root 1", "Root", alg1)
    r.addRole("Root 2", "Root", alg1)
    r.addRole("Root 3", "Root", alg1)
    r.addRole("Root 4", "Root", alg1)
    r.addRole("Root 5", "Root", alg1)
    r.addRole("Timestamp 1", "Timestamp", alg1)
    r.addRole("Snapshot 1", "Snapshot", alg1)
    r.addRole("Target 1", "Target", alg1)

    # Iterate through all dates in the range
    seq = EventList(start, end, seq_file)
    dates = seq.GetDates()
    events =  seq.GetEvents()

    for d in dates: # iterate through range of dates
        if d in events: # if this date matches an event
            if(verbose):
                print(f" - match {d.isoformat()}")
            r.doUpdate("Target 1") # require an update

        #for i in range(24):
        r.doTimestamp() # publish timestamp
```

```
    r.showStatsCSV()
    return()

def main():
    alg_list = SignatureAlgorithm.LoadFromCSV('algorithms.csv')

    for alg in alg_list:
        print(f"{alg}, ", end='')
        run_sequence(alg, False, "2019-01-01", "2045-01-01",
                     "device_A.csv", "Device_A")
        print(f"{alg}, ", end='')
        run_sequence(alg, False, "2020-01-01", "2046-01-01",
                     "device_B.csv", "Device_B")
        print(f"{alg}, ", end='')

if __name__ == "__main__":
    main()
```

E.4 Genetic Algorithm for Optimization

The optimization method for algorithm parameter selection uses the PyGAD package to build a genetic optimizer. This is linked to the simulator so that it can run multiple iterations with different parameter sets.

```
from datetime import datetime, timedelta
import pygad
import random
#import numpy

from TUFroles import *
from TUFsequence import *

start_date = "2019-01-01"
end_date = "2045-01-01"
sequence_file = "device_A.csv"
algorithm_file = "algorithms.csv"
device_name = "device_A"
roots = 1
swing = 0.0
alg_list = SignatureAlgorithm.LoadFromCSV("algorithms.csv")
seq = EventList(start_date, end_date, sequence_file)
dates = seq.GetDates()
events =  seq.GetEvents()

# Function to create a sequence of days between two defined dates
def date_range(start_date, end_date):
    for n in range(int((end_date - start_date).days) + 1):
        yield start_date + timedelta(n)
```

```
def run_sequence(sv):
    global swing
    global roots
    r = TUFRepository(device_name)

    for rn in range(1, roots):
        r.addRole("Root_"+str(rn), "Root", alg_list[sv[0]])
    r.addRole("Timestamp 1", "Timestamp",  alg_list[sv[1]])
    r.addRole("Snapshot 1", "Snapshot",  alg_list[sv[2]])
    r.addRole("Target 1", "Target",  alg_list[sv[3]])

    # Iterate through all dates in the range
    for d in dates:
        if d in events:
            r.doUpdate("Target 1")
            r.doTimestamp()
        else:
            r.doTimestamp()

    # r.showStatsCSV()
    cost = swing * r.getAccumulatedBytes() / 15000000 + \
           (1-swing) * r.getAccumulatedCost() / 32000

    fitness = 1.0 / abs(cost)
    return(fitness)


def show_sequence(sv, fit):
    global swing
    global roots
    r = TUFRepository(device_name)

    r.addRole("Root 1", "Root", alg_list[sv[0]])

    for rn in range(1, roots):
        r.addRole("Root_"+str(rn), "Root", alg_list[sv[0]])
    r.addRole("Timestamp 1", "Timestamp",  alg_list[sv[1]])
    r.addRole("Snapshot 1", "Snapshot",  alg_list[sv[2]])
    r.addRole("Target 1", "Target",  alg_list[sv[3]])

    # Iterate through all dates in the range
    for d in dates:
        if d in events:
            r.doUpdate("Target 1")
            r.doTimestamp()
        else:
            r.doTimestamp()

    print(f"{roots}, {swing}, {fit}, {alg_list[sv[0]]}, " +
          "{alg_list[sv[1]]}, {alg_list[sv[2]]},
          {alg_list[sv[3]]}, ",
          end='')
    r.showStatsCSV()
```

```
def fitness_func(ga_instance, solution, solution_idx):
    return run_sequence(solution)

def do_ga():
    global swing
    global roots
    swing = random.random()
    function_inputs = [32, 32, 32, 32]

    num_genes = len(function_inputs)
    gene_type=int
    init_range_low = 0
    init_range_high = len(alg_list) - 2

    fitness_function = fitness_func

    num_generations = 250
    num_parents_mating = 20
    sol_per_pop = 40
    keep_elitism = 5

    parent_selection_type = "rank"
    #parent_selection_type = "tournament"
    keep_parents = 2

    #crossover_type = "single_point"
    #crossover_type = "uniform"
    crossover_type = "two_points"

    mutation_type = "random"
    mutation_num_genes = 2

    save_best_solutions = True
    save_solutions = True

    ga_instance = pygad.GA(num_generations=num_generations,
                           num_parents_mating=num_parents_mating,
                           fitness_func=fitness_function,
                           sol_per_pop=sol_per_pop,
                           keep_elitism=keep_elitism,
                           num_genes=num_genes,
                           gene_type=gene_type,
                           init_range_low=init_range_low,
                           init_range_high=init_range_high,
                           parent_selection_type=
                             parent_selection_type,
                           keep_parents=keep_parents,
                           crossover_type=crossover_type,
                           mutation_type=mutation_type,
                           mutation_num_genes=
                             mutation_num_genes,
                           delay_after_gen=0,
                           save_best_solutions=
```

```
                              save_best_solutions,
                             save_solutions=save_solutions,
                             suppress_warnings=True)

    ga_instance.run()
    solution, solution_fitness, solution_idx =
      ga_instance.best_solution()
    show_sequence(solution, solution_fitness)


def main():
    global roots

    print("Roots, Swing, Fit, Root_alg, Timestamp_alg, " +
          "Snapshot_alg, " +
          "Target_alg, Device, Sig_Bytes, PK_Bytes, " +
          "Validation_Effort, " +
          "Total_Signatures")

    for roots in [1, 5]:
        for gan in range(0, 50):
            do_ga()

if __name__ == "__main__":
    main()
```