3-22-2018

Expected Coverage (ExCov): A Proposal to Compare Fuzz Test Coverage within an Infinite Input Space

Evan V. Swihart

EXPECTED COVERAGE (EXCOV): A PROPOSAL TO COMPUTE FUZZ TEST COVERAGE WITHIN AN INFINITE INPUT SPACE

THESIS

Evan V. Swihart

AFIT-ENG-MS-18-M-063

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-063

EXPECTED COVERAGE (EXCOV): A PROPOSAL TO COMPUTE FUZZ TEST COVERAGE WITHIN AN INFINITE INPUT SPACE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Evan V. Swihart, B.S.E.E.

March 2018

AFIT-ENG-MS-18-M-063

EXPECTED COVERAGE (EXCOV): A PROPOSAL TO COMPUTE FUZZ TEST COVERAGE WITHIN AN INFINITE INPUT SPACE

THESIS

Evan V. Swihart, B.S.E.E.

Committee Membership:

Maj T. Carbino
Chair

Dr. S. Graham
Member

Dr. B. Mullins
Member

Mr. C. Sielski
Member

AFIT-ENG-MS-18-M-063

Abstract

A Fuzz test is an approach used to discover vulnerabilities by intentionally sending invalid inputs to a system for the purpose of triggering some type of fault or unintended effect that renders the system vulnerable to an exploit. Fuzz testing is an important cyber-testing technique used to find and fix vulnerabilities before they are exploited. The fuzzing of military data links presents a particular challenge because existing fuzzing tools cannot be easily applied to these systems. As a result, the tools and techniques used to fuzz these links vary widely in sophistication and effectiveness. Because of the infinite, or nearly infinite, number of possible fuzzed messages that can be sent on a military data link, measuring the coverage of a fuzz test is not straightforward. This thesis proposes an understandable and meaningful metric for protocol fuzz testing called ExCov. This metric computes the coverage of a fuzz test set from a probabilistic model of vulnerability occurrence and defines coverage as the expected percent of existing vulnerabilities discovered by a set of test cases. This metric enables the acquisitions community to more succinctly write weapons system requirements for cyber security. Furthermore, it quantifies the number of faults and vulnerabilities that are expected to be found by a set of test cases, which provides decision makers with valuable information to make more informed choices on whether or not to perform additional testing. As a result, industry will be better equipped to determine cost and effort when performing cyber vulnerability testing. In addition, industry will also be able to more concretely represent the results of the cyber testing they perform. ExCov was implemented in a suite of tools called ExFuzz, and these tools were used to compare and contrast military data link fuzz testing techniques that are in use today. By assessing these current methods using the ExCov metric,

optimal bit flip probabilities for the mutative fuzzing of three custom protocols was found. A generative fuzzer was also built based on the metric and was shown to outperform mutative and manual generation strategies in nearly every case.

Table of Contents

Page

List of Figures

Figure Page

List of Tables

EXPECTED COVERAGE (EXCOV): A PROPOSAL TO COMPUTE FUZZ TEST COVERAGE WITHIN AN INFINITE INPUT SPACE

I. Introduction

Vulnerability discovery is a key task when securing any type of digital system. When that system is a vehicle, aircraft, or missile, the implications of a malicious actor discovering and exploiting a vulnerability are grave. Security researchers must identify and fix these vulnerabilities before a bad actor finds and exploits them.

The development of effective and efficient methods of vulnerability discovery is an important step in the effort to secure all types of embedded systems. One approach to vulnerability discovery involves sending intentionally invalid inputs to a system in the hopes of triggering some type of fault or unintended effect that may be render the system vulnerable to an exploit. This type of invalid input testing is called fuzz testing, or fuzzing.

1.1 Redefining the Air-gap for Our Weapon Systems

Effective negative testing is an important part of cyber defense. In embedded weapon systems, data flows between many different sub-systems, using a myriad of different protocols. Fuzz testing can be used to identify messages that, if sent between these sub-systems, may impact a system's performance or compromise a system entirely. The following paper, [23], motivates the need more for negative testing, including effective fuzz testing, from a deterrence policy perspective. The paper was co-authored by Evan Swihart and Lt Col Mark Reith and was accepted to the March 2018 International Conference on Cyber Warfare and Security.

Redefining the Air-gap for Our Weapon Systems

Evan Swihart, Lt Col Mark Reith

1. The threat

There are many reasons the adversary may want to look beyond the traditional battlefield of TCP/IP enterprise networks and target the cyberspace of our weapon systems. Two major reasons are the low cost, both financial and political, and the high impact of such an attack.

The cost to stand up programs to design, build, and deploy systems capable of kinetically destroying air assets can easily be in the billions of dollars (National Priorities Project, 2017). While it is difficult to estimate the cost of standing up a cyber program capable of incapacitating our air systems, it is certainly orders of magnitude less. Cyber exploits are also politically cheap. Since cyber-attack attribution is difficult, perpetrators can hide behind a veil of plausible deniability. When accused of an attack, the adversary can claim to have nothing to do with it and avoid international and domestic political consequences for the action. Prominent likely state sponsored cyber-attacks such as Stuxnet (Kushner, 2013), the Ukrainian Power Plant hack (Zetter, 2016), and the Sony hack (Zetter, 2014), all have the common theme of no state claiming responsibility. If a cyber-attack targets our weapon systems, we can expect attribution to be difficult and any retaliation attempt to be complicated. In contrast, attribution of a kinetic attack against our weapon systems will be simple, and the perpetrators will likely face a kinetic response.

The scope of the damage that could be caused by a cyber-attack on our weapon systems may also entice an adversary. A kinetic weapon is designed to destroy assets one at a time, and destroying a single asset has little effect on the other assets. Cyber weapons operate differently. They are designed to target a vulnerability that exists

in all systems of the same or similar configuration. Therefore, these weapons can be targeted at a type of system rather than a single asset. They can be delivered in a variety of ways including a supply chain attack (Shackleford, 2015), where modified parts find their way into each asset, or the compromising of equipment on the flight line which will touch many aircraft. Thus, a single cyber weapon could incapacitate an entire type of weapon system.

2. Our weakness

We may not be prepared to treat our systems as contested cyberspace. Our current approach to acquisitions and weapon system design assumes our systems are free of adversarial interference. This assumption magnifies the effect a successfully delivered cyber weapon would have on our systems. For example, when an avionics box is added to a data bus on an aircraft, we design and test the box to function properly with everything else on that bus. We often do not design for or test what would happen if unexpected and invalid messages are sent to our new box. Should one terminal on a data bus be compromised such that it sends invalid messages out on the bus, we may have no idea how every receiving terminal would react. Similarly we often trust other friendly systems external to our weapon systems such as navigational aids like GPS, tactical data links, and mission and key loading equipment to be free of adversarial interference. We often do not design or test for the case that they are not.

It is entirely understandable why this assumption of trust has prevailed in our thinking thus far. It is a simple and logical conclusion that since these weapon systems are not physically connected to the Internet or the DoD Information Network (DoDIN), the adversary can not cause any cyber effects on a platform without physical access. In fact, weapon systems are specifically excluded from the definition of the DoDIN (Defense Acquisition University, 2017). This notion of security by air-gap is flawed (Guri et al, 2014). Reliance on data and computer networks for weapon

system operations means that exchanges of information across the air-gap are common and even required for functionality. Information can reach an air-gapped weapon system over RF, through maintenance and test equipment, through mission loads, via attachable hardware like weapons and other pods, or even in counterfeit parts slipped into the supply chain. Malicious information can find its way into a weapon system through any of these channels.

Our requirements writing and testing practices reflect our misplaced trust in the integrity of our systems. We write requirements for systems to correctly receive and process valid inputs. We then test that the system that is delivered meets these requirements. This is considered positive testing as it tests if the system handles valid inputs as it should. Since we trust other systems to send only valid inputs, there has been no reason to write requirements for systems to handle invalid inputs. And since we only test that the delivered system meets its requirements, we generally do not test invalid inputs. If we did test invalid inputs, we would call this testing negative testing. Negative testing is crucial to understanding how a cyber-attack would affect a weapon system. In conventional networks, input validation is a high priority security control (National Institute of Standards and Technology, 2013); it should be in embedded systems as well. Without any requirements or testing aimed at preventing cyber effects from propagating throughout our systems, there is no way we can defend this space.

3. Rethinking the air-gap

It is clear that the acquisitions process for our weapons systems will need to be reformed to adapt to the threat of cyber warfare in our weapon systems. In response to this threat, the Air Force Material Command has started the Air Force Cyber Campaign Plan which aims to address this issue through seven lines of attack (LOAs) (Gross, 2016). The recommendations in this paper can, and in some cases are, being

implemented under LOAs 1, 2, 4 and 6.

We can deter the adversary from contesting the cyberspace of our weapon systems by designing and testing systems that make cyber exploitation very difficult, unreliable and costly. This can be done by focusing on two key elements that make cyber exploits possible, an access vector to the system and persistence and mobility once inside the system. If we successfully deter the adversary from contesting the cyberspace of our weapon systems, we can effectively consider these systems to be air-gapped. That is, we can consider the cyber space of our weapon systems to be distinctly separate and secure from the DoDIN and Internet.

3.1 Preventing access

There are two key actions we can take to prevent the adversary from accessing our weapon systems. First, we can write requirements that require the proper handling of invalid inputs from external sources, and test that these requirements have been met. This will reduce the number of flaws in our systems that could be exploited by an adversary to gain access. Many systems handle data coming and going from a platform before during and after a mission. A key access vector is exploiting flaws in the protocol or implementation of these data links to move malicious information to the air platform. This can be countered by recognizing that, unlike in the TCP/IP enterprise world, the adversary will have limited access to these peripheral sub-systems to test their cyber weapons. Since we do have access to them, a small bit of negative testing of the most vulnerable message types would reveal the most likely exploits at a fraction of the investment the adversary would need to make to discover them. For new and upgraded systems, requirements should be added to ensure that such invalid inputs are gracefully handled by the system. Fixing these vulnerabilities would vastly increase the sophistication of the cyber weapon the adversary would need to compromise a weapon system. With limited access to the system, creating

this weapon would prove a difficult task for the adversary.

It is worth pointing out that this argument relies heavily on the discredited idea of security by obscurity (Open Web Application Security Project, 2017). The idea of security by obscurity in computer networks is flawed because, as it turns out, when more people look at a system and find flaws, many act responsibly and report the flaws, thus a more secure system can be built. This logic does not apply to weapon systems. The only people with the means and will to test these systems, even if component designs were public, are nation states. For these reasons, only our adversaries would benefit from the public release of our weapon systems designs. Therefore, security by obscurity is a valid security method in this context.

Second, for other types of access to the platforms systems such as data loads, attestation may make it far more difficult for the adversary to gain access to a system (Seshadri et al, 2004). Attestation is a tool used to verify that data has not been modified by comparing values in memory to a trusted copy. If we implement attestation at key points where instructions to the weapon system are inputted, we may be able to detect moderately sophisticated attempts to force the system to perform in an unexpected way. Attestation can be used to verify that one of a discrete and proven set of valid instructions is loaded to ensure no malicious instructions slip onto our systems. It can also help prevent supply chain attacks by ensuring malicious information is not present in a system before it is installed by cross checking software loads against proven versions.

3.2 Preventing persistence and mobility

In addition to making access to our systems more difficult and expensive for our adversaries, we can also mitigate the effects of their cyber weapons should they gain access. This will not only reduce the effect of cyber weapons used against our embedded weapon systems, but also reduce the incentive for adversarial action.

Attestation can also be used for this purpose by preventing malicious code from persisting in our weapon systems. By regularly comparing the software loaded in systems on the aircraft with proven copies elsewhere, illegal modifications can be detected. This will make it harder for the adversary to persist if they find a way to access a weapon system.

One way of preventing malicious information from spreading through a weapon system is to use simple data transfer schemes inside the platform that limit the possible data each element in the system can transmit or receive. By limiting the transferable data, the variety of inputs each system may receive is reduced. This will make it much harder to transfer bad or malicious data throughout the system. In TCP/IP Enterprise networks, computers are usually general purpose, so limiting their ability to pass types of data would be detrimental to their performance. In embedded systems however, the elements are specialized, and limiting their inputs and outputs will not greatly affect their operation. There are a number of efforts to standardize the information passed between components in a weapon system. They include Open Mission Systems (Virtual Distributed Laboratory, 2017) and Future Airborne Capability Environment (The Open Group, 2017).

Another way to prevent the spread of malicious information inside a weapon system is to once again require systems to properly handle invalid inputs. This will reduce the risk that an invalid message sent from one trusted, compromised subsystem to another uncompromised system will cause the uncompromised system any problems. Rigorous negative testing on data busses inside our systems can reveal these types of vulnerabilities and correcting them will make it very difficult for cyber effects to spread through our systems.

3.3 Preventing persistence and mobility

So far this paper has discussed security solutions that focus on improving our

design, our testing and our verification of data before and after missions. A natural question to ask is, Should we do more to actively detect cyber intrusions into our weapon systems? In computer networks, intrusion detection systems (IDS) do just this (Rowland, 2002). Research has been done that extends IDSs into embedded systems with some success (Tabrizi and Pattabiraman, 2015). An IDS on an aircraft has even been proposed (Buehler and Duffner, 2017). However there are two fundamental requirements for IDSs to be effective that may be difficult to realize in the environment of our embedded weapon systems. Even if we can support these requirements, the nature of our weapon systems suggests that IDSs will not be very effective in this domain.

First, IDSs require some space, physical or digital, to reside. A network based IDS would need wired connections to all the different busses on an aircraft and a place to reside that could accommodate the necessary data ports. Given the size weight and power limitations on aircraft, this could prove challenging to accommodate. In addition to this, unlike common lower layer protocols that make up computer networks like Ethernet and 802.11, military data busses may not easily accommodate interloping devices. Busses running at different levels of classification also presents a challenge to network based IDSs. An alternative to network based IDSs is host based IDSs where the software resides on hosts instead of in its own space (Tabrizi and Pattabiraman, 2015). This solves some of the problems of network IDSs, but requires every piece of avionics to be updated with this new software. Surely it would be easier and cheaper to update the avionics to simply not process any messages an IDS would discover, rather than implement an IDS on top of the equipment.

The second requirement for an IDS is some sort of database that allows the software to detect malicious traffic. To build this database, aircraft systems would need to be studied in many different environments to construct a profile of valid and invalid

traffic. If this study is completed, we might as well use the information to update systems to accept and process data that meets the valid profile rather than implement a separate IDS system. IDS in traditional computer networks makes sense because traffic unusual for that specific network can be detected and stopped without compromising the general purpose nature of the hosts. Nothing is general purpose in avionics however, so it makes sense to limit what hosts receive instead of implementing an IDS.

4. Deterring the adversary

Implementation of these practices will not be cheap. Investments in additional testing, the development and deployment of attestation, and new requirements on input checking are sure to drive up the acquisition costs of our weapon systems. However, making this investment will force the adversary to either invest more heavily to defeat our cyber defenses, or to revert to opposing our weapons-systems by conventional, kinetic means. The embedded system nature of our weapon systems means that our investment in security would require a much larger investment by the adversary to overcome.

Since these systems are often designed for use only by the DoD and trusted allies, the cost to the adversary to acquire them to identify vulnerabilities is higher than our cost to acquire them for testing. Also, since the data flowing on an aircrafts busses can be completely specified and standardized, the aircrafts systems can be designed to drop invalid messages and receive and transmit only valid information. This degrades the ability of the adversary to affect a secure subsystem from a compromised subsystem and thus limits the affect an adversary can have.

With reduced ability to access the cyberspace of weapon systems, and virtually no ability to spread malicious information once inside a weapon system, the cost benefit calculation of the adversary would shift and incentivize vacating this cyber battlefield. In effect, the concept of an air-gap would be restored and the weapon systems would

be removed from contested cyberspace. With vigilance, good security practices, and testing, we can prevent the adversary from deciding it will be worth their effort to contest the cyber space of our weapon systems ever again.

5. Conclusion

Extending cyber war to the domain of our embedded weapon systems is a low-risk high reward venture for our adversaries. We are currently at a disadvantage while defending this terrain because our approach to weapon system acquisition and design supposes these systems are free of adversarial interference. We risk losing our ability to project conventional air power should an adversary sufficiently disrupt these systems. However, due to the unique design of embedded weapon systems, there is an opportunity to deter the adversary from contesting this space with key investments in negative testing, requirements to block invalid and potentially malicious data, and promising technologies such as attestation for embedded systems. These investments would deter the adversary from contesting the cyberspace of our weapon systems, providing the effect of an air-gap between them and the cyber battlefields of the DoD Information Network and larger Internet.

Works Cited

- Buehler E. and Duffner, K. (2017) Avionics intrusion detection system and method of determining intrusion of an avionics component or system, US Patent, 9,591,005.
- Defense Acquisition University, Glossary of Defense Acquisition Acronyms and Terms, accessed 25 July 2017 Department of Defense Information Network (DoDIN), [online], https://dap.dau.mil/glossary/pages/3348.aspx.
- Gross, C. (2016) AFMC commander says cyber threats are real, need to get ahead of them, [online], US Air Force News Service,

http://www.af.mil/News/Article-Display/Article/951715/afmc-commander-says-cyber-threats-are-real-need-to-get-ahead-of-them/.

- Guri, M., Kedma, G., Kachlon, A. and Elovici, Y. (2014) AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies, 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE), pp. 5867.
- Kushner, D. (2013) The Real Story of Stuxnet, [online], IEEE Spectrum, http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet.
- National Institute of Standards and Technology, (2013) Special Publication 800-53 (Rev. 4): Security Controls and Assessment Procedures for Federal Information Systems and Organizations, pp. F-229-230.
- National Priorities Project, accessed 28 July 2017 Analysis of the Fiscal Year 2012 Pentagon Spending Request, [online], https://www.nationalpriorities.org/analysis/2011/analysis-fiscal-year-2012-pentagon-spending-request/?redirect=cow.
- Open Web Application Security Project, accessed 17 August 2017 Security Design Principles, [online], https://www.owasp.org/index.php/Security_by_Design_Principles.
- Rowland, C. (2002) Intrusion detection system, US Patent, 6,405,318.
- Seshadri, A., Perrig, A., Van Doorn, L. and Khosla, P. (2004) SWATT: software-based attestation for embedded devices, IEEE Symposium on Security and Privacy, 2004, Proceedings, pp. 272282.
- Shackleford, D. (2015) Combatting Cyber Risks in the Supply Chain, SANS Institute.
- Tabrizi, F.M. and Pattabiraman, K. (2015) Intrusion Detection System for Embedded Systems, Proceedings of the Doctoral Symposium of the 16th International Middleware Conference on - Middleware Doct Symposium 15, pp. 14.
- The Open Group, accessed 26 July 2017 About the FACE Consortium, [online], http://www.opengroup.org/face/about.
- Virtual Distributed Laboratory, accessed 26 July 2017 Open Mission Systems (OMS), [online], Air Force Research Laboratory, https://www.vdl.afrl.af.mil/programs/uci/oms.php.

- Zetter, K. (2014) The Evidence that North Korea Hacked Sony is Flimsy, [online], Wired, https://www.wired.com/2014/12/evidence-of-north-korea-hack-is-thin/.
- Zetter, K. (2016) Everything We Know About Ukraines Power Plant Hack, [online], Wired, https://www.wired.com/2016/01/everything-we-know-about-ukraines-power-plant-hack/

1.2 Motivation for Thesis

The preceding paper calls for rigorous negative testing. It states, "Rigorous negative testing on data busses inside our systems can reveal ... vulnerabilities and correcting them will make it very difficult for cyber effects to spread through our systems." [23] This thesis advances the negative testing technique of fuzzing by presenting a tool that improves test case selection compared to the tools and methods in use by the Air Force today.

The paper also states that the Air Force must "Write requirements that require the proper handling of invalid inputs" and "test that these requirements have been met". [23] To put such a requirement on contract, the Air Force needs to specify a testing approach that can determine if invalid inputs are being handled properly. Every possible invalid input cannot feasibly be tested, so it is important that the testing approach provides high coverage to give testers confidence that if vulnerabilities exist, they will be found.

Currently there is no way to quantitatively compare testing approaches, so it is hard to say with confidence which approaches provide high coverage. This thesis proposes a method for measuring the coverage of a fuzz test, thereby making it possible to compare testing approaches. The proposed method also provides meaning to the coverage metric, so that if a fuzz test provides 50% coverage, it can be said that the test expects to uncover 50% of all existing vulnerabilities. This information could allow for requirements that let the contractor decide the testing method and instead require their system to be, for example, "fuzz tested to 90% coverage." Not only does this allow the contractor flexibility to meet the requirement, but it provides risk managers with an indication of the probability that vulnerabilities remain in the system.

1.3 Thesis Overview

Broadly speaking, vulnerability discovery efforts can be placed into three categories based on the information available to the security tester: white box, gray box, and black box testing. In white box testing, the tester has complete access to information about the System Under Test (SUT). In black box testing the tester has little or no information about the inner workings of the SUT, and can only observe the outputs of the system for any applied input. Gray box testing is somewhere between black and white box testing. Often gray box testing is associated with having access to compiled binaries and some documentation and involves some reverse engineering as well as black and white box testing techniques.

This thesis focuses on a technique used in black and gray box testing called fuzzing. According to [22], "Fuzzing is the the process of sending intentionally invalid data to a product in the hopes of triggering an error condition of fault." More specifically, this thesis focuses on protocol fuzzing as opposed to other types of fuzzing such as file format fuzzing or web application fuzzing. Protocol fuzzing involves creating input messages that do not meet the requirements of a protocol specification or standard, injecting the bad messages onto a link, and then assessing the receiving system for faults or unexpected behavior that may indicate a vulnerability is present.

The main focus of the thesis is the generation of input messages for this process. It provides the following four contributions:

1. A coverage criterion and coverage calculator that quantitatively assess the effectiveness of a fuzz test. Section 1.5 introduces the basis for this criterion, Chapter III is devoted to the development of this criterion and Section 4.2 discusses the implementation of the criterion in the form of a coverage calculator application.

2. A method of representing a protocol specification in an XML file called a data model, and an application that converts this file to a C++ object for use by other applications. Section 4.1 covers this method and application.

3. A generative fuzzing application that automatically generates a high coverage test set based on a data model. This application is covered in Section 4.3.

4. A comparison of the coverage provided by a number of fuzz test sets that are generated in a variety of ways. Chapter V is devoted to this topic.

1.4 Fuzzing Framework

A fuzz test consists of many different components. The tester needs a way to create a set of fuzzed inputs, a method to assess the effectiveness of the test set, a way to deliver the messages, and a means of assessing the messages' impacts on the SUT. A fuzzing framework can be used to encapsulate all of these aspects in a single process. Figure 1 illustrates a general fuzzing framework and shows the input output

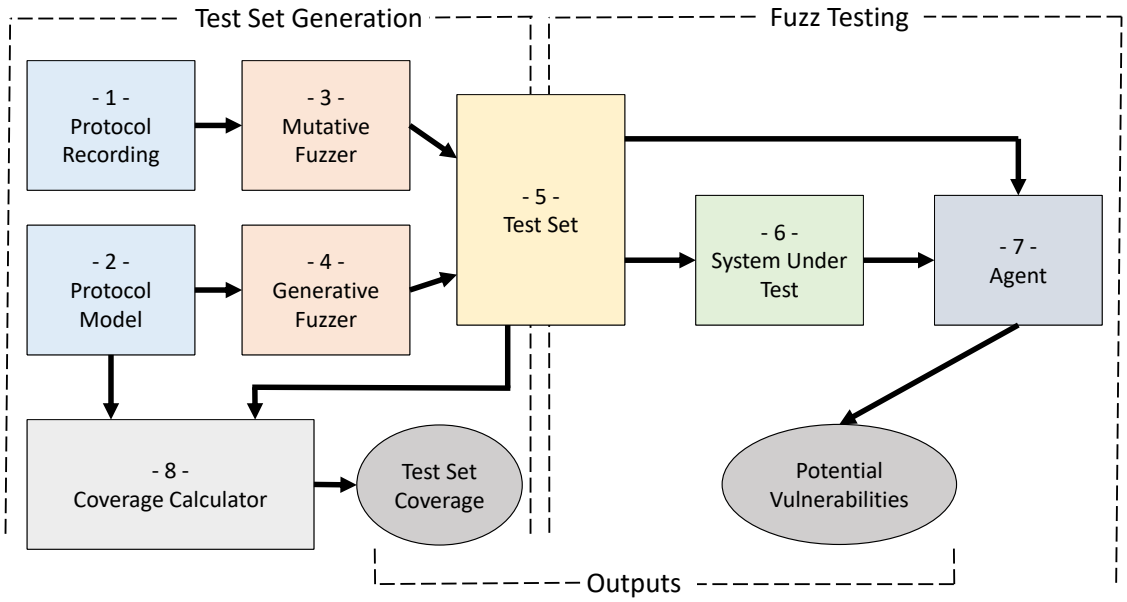

Figure 1. An Overview of Protocol Fuzz Testing

relations between components in the test. The figure was partially based on an image found in [5]. The following list explains each component and its role in the thesis:

1. **Protocol Recording:** A recording of valid data being passed to the system under test. These recordings can be mutated by a mutative fuzzer to generate fuzz test cases. This thesis simulates protocol recordings by generating them in specially designed spreadsheet tools. This method of fuzz testing is compared to others in Chapter V.

2. **Protocol Model:** A model of the protocol specification that defines the meaning of binary data in a data link. A generative fuzzer requires such a model to build fuzz test cases. This model is also required to compute coverage. It is in this model that a tester can convey knowledge of potentially vulnerable fields or values. For this thesis, a protocol model is also called a data model and takes the form of an XML document. The structure of this document is a modified version of the Peach Pit format used by the fuzzing platform, Peach [10].

3. **Mutative Fuzzer:** A mutative fuzzer operates on a recording of valid traffic of the protocol to be fuzzed. It mutates various aspects of these recordings to make them invalid. The output of a mutative fuzzer is a set of test cases to be tested against a system. If a data model for the protocol exists, the coverage of these test cases can be calculated. In this thesis, the coverage of the test sets generated by mutative fuzzers will be compared to those generated by generative fuzzers in Chapter V.

4. **Generative Fuzzer:** A generative fuzzer takes the specification of the protocol, or the data model, and builds test cases from it. This thesis involves building a generative fuzzer, discussed in Section 4.3, that provides high coverage based on the coverage criterion described in Chapter III. The generative fuzzing approach

is compared to other approaches in Chapter V.

5. **Test Set:** A set of test cases to be applied to the system under test. These messages can be generated by a mutative or generative fuzzer, or can even be crafted manually. The coverage these test cases provide can be computed using the coverage criterion described in Chapter III with knowledge of the protocol model. Knowledge of the coverage of the test set can help the tester make decisions about further testing and risk management.

6. **System Under Test:** The SUT is the hardware and software that is being fuzzed in the fuzz test. In this thesis, the SUT primarily being considered is a military data link radio and the other hardware and software systems that process the data received from the link.

7. **Agent:** The agent is responsible for monitoring the SUT and detecting if a vulnerability has been discovered. If so, the agent correlates the vulnerability with the input that caused it. The agent can simply be a human monitoring the system as the test progresses, but if the agent is automated, the test may be able to run much faster. An automated agent also allows for a real-time feedback loop from the agent to the test set generation element. This allows for "smart fuzzing" where the test sets being applied are selected based on the effect of previous tests. Unfortunately for the military data link systems under consideration here, the creation of an automated agent is a major challenge. Since this thesis focuses on the test set generation side of the framework, the agent will not be discussed further in this thesis.

8. **Coverage Calculator:** The coverage calculator is an application that takes a protocol model and a set of test cases and returns the coverage percentage of those test cases based on the coverage criterion defined in Chapter III. For this

thesis, a coverage calculator was created to demonstrate the feasibility of the proposed coverage criterion. It is also used to compare different approaches to test set generation in Chapter V.

Using this framework, at the conclusion of a fuzz test the tester will have not only a list of potential vulnerabilities, but also a metric for the effectiveness of the test. This metric will allow the tester and user of the system to understand the likelihood that the test missed a vulnerability, and base further testing and operational decisions on that knowledge. It is a definition of this metric that is the primary contribution of this thesis.

1.5 Assessing Fuzz Tests

There is effectively an infinite number of test cases that can be applied to any major system. Consider the navigation message of the Global Positioning System — Standard Positioning Service (GPS-SPS). One sub-frame of this message is 300 bits long and takes 6 seconds to transmit [4]. This means that there are $2^{300} = 2.04 \times 10^{90}$ possible sub-frames. At 6 seconds a piece it would take 1.22×10^{91} seconds or 3.87×10^{83} years to transmit every possibility. Considering the universe is a mere 1.38×10^{10} years old, such a test length is unacceptable. This is all before considering that the GPS-SPS message consist of 5 sub-frames, some of which are interpreted differently depending on when they are sent.

Since testing every possible input is not feasible, the tester must select a subset of possible inputs to test. There is no obvious approach to selecting these inputs. The tester, having some experience with the SUT, may have some inclinations about inputs that could prove problimatic. And some inputs, like valid ones, would not contribute to the fuzz test at all. The tester could manually create a test set based on expertise with the system. This would be time consuming, but provide a short

test that covers the most crucial inputs. A test set could also be generated randomly by a computer. This method could quickly create many test cases, but their quality wouldn't be as good, relying on chance to generate the most suspicious inputs.

A further challenge is assessing the quality of a test set. The term "coverage" is often used as an intuitive descriptor of the effectiveness of a test set, but this term has yet to be precisely defined. A test set manually crafted by an expert with knowledge of the protocol is bound to be more effective, and thus provide more coverage, than a randomly generated set of the same size. But how would testers compare a manually crafted set of 100 inputs to a randomly generated set of 10,000? Does the increased number of randomly generated test cases make up for the lack of insight used in their generation? What if the tester built a tool that automatically generated 1000 inputs based on the most vulnerable areas of the protocol? How would its coverage compare?

The main reason these questions are difficult to answer is that there is no quantitative measure for the effectiveness, or coverage, of a fuzz test. In the literature, the measure of effectiveness, or adequacy, of a test is called an adequacy criterion or testing criterion [11], [29]. Colloquially, the effectiveness of a test as the test's coverage, and the method of measuring this coverage is reffered to as the coverage criterion.

Software testing coverage criterion have been well researched and applied in the past few decades, however these criterion are rarely applicable to fuzz testing. There are two attempts to define a fuzzing coverage criterion in the literature. One, [14], defines coverage based on state transitions, and the other [26], defines it based on tester defined constraints extracted from the protocol specification. Both of these approaches have advantages, but neither defines coverage in a way consistent with what the term suggests. That is, if these criterions return a coverage of 100%, they may still miss some vulnerabilities. Also, the meaning of any coverage value found

based on these criterions requires an in depth understanding of what the criterion is measuring exactly. They, and other coverage criterion approaches are discussed in Chapter II.

In this thesis a new definition of coverage is proposed that is rooted in providing an understandable metric that stands for something meaningful. By using such a metric, a tester can make accurate assessments of the effectiveness of different methods of fuzzed test set generation. Furthermore, a tester can make informed decisions about how much to test, and can accurately assess the risk that a test missed vulnerability. Since the goal of a fuzz test is to uncover vulnerabilities, it should be measured based on its ability to do just that. Therefore this thesis proposes a coverage criterion with the term coverage defined as *the expected percent of existing vulnerabilities discovered by a set of test cases*.

This metric is not simple to calculate, but has real meaning if calculated successfully. For example, if a fuzz test is found to have a coverage of 90%, a tester can say with confidence that, based on the information available, this test is expected to uncover 90% of the existing vulnerabilities. Perhaps there is a more time consuming and expensive test that could increase coverage to 95%. If the tester knows that the first test found 9 vulnerabilities, it can be said that the second test expects to find 0.5 more vulnerabilities. Now the decision of whether or not to perform the more expensive test can be made based on this quantitative information.

For this metric to be successful, it must be able to incorporate everything the tester knows about a protocol and the nature of vulnerabilities in general. The metric cannot possibly provide the exact number of vulnerabilities a test will uncover, that would require knowing them ahead of the test, but it can provide a probabilistic estimate based on all the information available. The more information about a protocol and about how vulnerabilities manifest in protocols that is incorporated into the criterion,

the more accurate the criterion can be.

Chapter III derives the equations necessary to compute the coverage of a set of test cases based on our definition of coverage. To do this, it makes some assumptions about the nature of vulnerabilities, but it is designed to be adaptable to new information that changes these assumptions. The method also allows for tester input about features of the protocol that will influence the coverage metric. For example, known or suspected vulnerable inputs can be supplied to the coverage calculator so that the increased likelihood a vulnerability is present there is reflected in the metric.

In order to compute the coverage of any test set, a model for the associated protocol must be created. The structure of this model, called a data model, is based off the data model used in an existing commercial fuzzing framework. This model carries all the information about the protocol structure and potentially vulnerable inputs that the coverage calculator requires to compute an accurate coverage value. The creation of this model is discussed in Chapter IV.

The newly created coverage criterion was implemented in a coverage calculator as part of this thesis. The calculator demonstrates that the criterion is implementable, and in Chapter V, the calculator is used to compare test set generation techniques and to provide answers to some of the questions posed earlier such as: does 100 manually crafted inputs or 1000 randomly generated cases provide more coverage?

II. Related Work

In this chapter, the history of fuzzing will be briefly discussed as well as a snapshot of the current state of the art in protocol fuzz testing. From there the chapter considers two prominent fuzzing frameworks and explains how they influenced elements of this thesis, and how they may or may not be applied to military data link systems. The final section discusses the history of software test assessment and explains the development of the coverage criterion concept. This section also considers, in-depth, two coverage criterion for fuzz testing and explains why the criterion proposed in this thesis is a necessary addition.

2.1 Fuzz Testing

2.1.1 In The Literature.

The first mention of fuzzing in the scientific literature dates back to 1990 when a study sought to test the reliability of UNIX applications [18]. The researchers supplied random character inputs to a number of applications and monitored for crashes. The program they used to generate the random characters was called fuzz. Little more was done in the decade until researchers at the university of Oulu began work on a test suite called PROTOS in 1999 [13]. This set of tools was capable of delivering packets that did not meet the proper protocol specifications. These malformed packets could be used to test the robustness of network interfaces and expose implementation errors.

Shortly thereafter Dave Aitel introduced an open source fuzzer called SPIKE [8]. SPIKE allowed users to model network protocols with variable length data blocks, making it easier to fuzz all levels of the protocol stack. SPIKE, and the block-based approach to protocol modeling, became the foundation for the modern fuzzing tool Sulley which is discussed later in this chapter.

In the early 2000s the technique was extended from UNIX application fuzzing and network protocol fuzzing to other types of application fuzzing. Web browser fuzzing gained attention in 2004 when Michal Zalewski released a tool called mangleme [22], [28]. The tool generated randomly mutated HTML pages to expose vulnerabilities in web browsers. In 2005, Michael Sutton and Adam Greene from iDEFENSE labs presented a variety of file fuzzing tools at Black Hat USA in 2005 [21]. These tools allowed users to automatically generate corrupt files that could expose vulnerabilities in the applications that use them.

In the 2010s protocol fuzzing research has focused on developing tools that leverage as much information as possible to generate fuzz sets that cover as much of a protocol as they can, or are simple to implement and require little human effort to operate. Despite tools tailored for many different applications and industries, little or no published fuzzing research is tailored to military data link systems. This is not surprising given the sensitive nature of such systems. There are however a variety of tools that, while they may not be directly useful, illustrate that types of innovative fuzzing implementations that are feasible in similar systems. It is important to have a good understanding of the state of the art in this regard so that testers can envision the types of tools that may be designed and employed for military data link systems. Three such innovative tools are described here:

- AutoFuzz is a type of mutative fuzzer called an in-line fuzzer. This type of fuzzer sits between two communicating entities and mutates some messages flowing between them in real time. In the case of AutoFuzz, the fuzzer sits between a server and client. The tool learns the syntax of the protocol automatically by observing valid messages pass between the server and client for a period of time. At some point it begins to apply mutations to packets passing through the system, fuzzing either the server or the client. An illustration of this system

is shown in Figure 2. This type of self-learning, plug and play fuzzing tool is an intriguing concept that may also be implementable for military data link systems. [12]

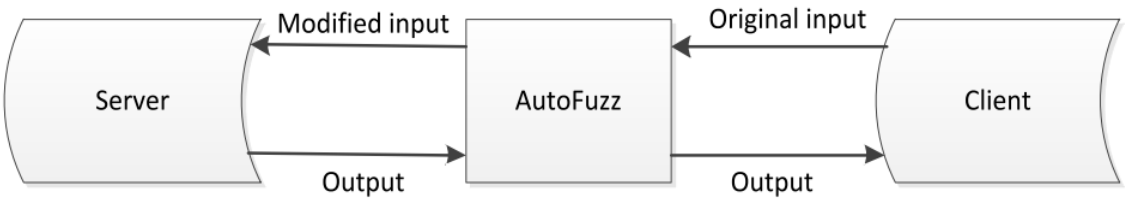

Figure 2. The set-up of the in-line fuzzer, AutoFuzz. This image was retrieved from [12].

- T-Fuzz is a generative fuzzer designed for use in telecommunication networks. The fuzzer is integrated into a conformance testing framework that is already in use in the industry. This existing framework happens to include protocol models; T-Fuzz leverages this fact by extracting these existing models and producing fuzzed messages based on them. The fuzzer is also integrated with the existing test delivery mechanism. A diagram of T-Fuzz and its interaction with the conformance testing tool TTCN-3 is shown in Figure 3. The approach taken by T-Fuzz shows how a fuzz testing tool can be integrated into an existing conformance testing tool. The approach also avoids the tedious task of protocol modeling by extracting existing models from the testing framework. [16]

- Another fuzzing approach involves leveraging research into taint analysis to direct fuzz testing towards the most vulnerable network data. Taint analysis is a program analysis technique that tracks user input data throughout a program and system; the idea being that this data is a likely source of unwanted inputs, or taint. This fuzzing approach tracks tainted packets and data in a network and fuzzes them. By targeting this type of data, the researchers hope to more quickly identify vulnerabilities in a network. This type of targeted fuzzing

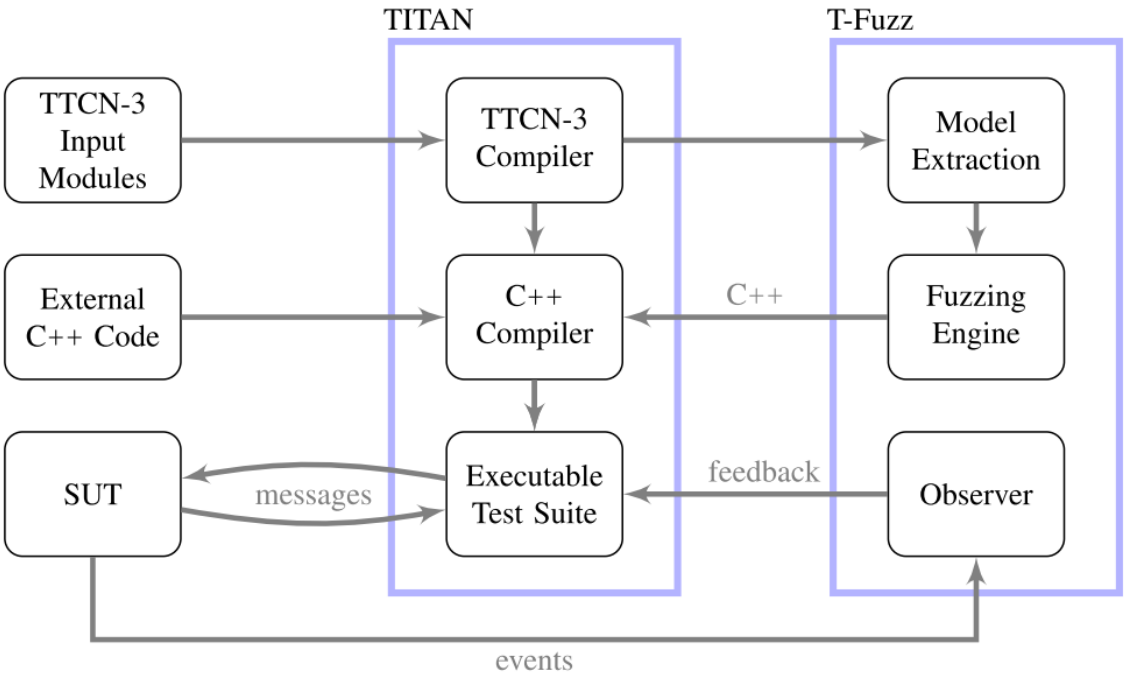

Figure 3. The set-up of the generative fuzzer, T-Fuzz. This image was retrieved from [16].

can be envisioned in military data link systems. For example, if information from untrustworthy external sources could be tracked and exclusively fuzzed, high risk susceptibilities may be found more quickly than if fuzzing is applied uniformly. [9]

2.1.2 In Practice.

While fuzzing frameworks specific to military data links do not appear in the literature, general purpose frameworks offer an opportunity to use existing tools to fuzz such links. A good overview of general purpose tools can be found in a report on securing critical infrastructure commissioned by the European Union. The report, [5], is focused in part on the protocol fuzzing of industrial control systems, often embedded systems.

Industrial control systems face some of the same challenges as military data link systems when it comes to fuzz testing. First of all, both are predominantly embedded

systems that a built for specific tasks. This means that they lack some of the general purpose features of traditional computer networks. As a result, it is far more difficult to apply traditional fuzzing tools and error monitoring applications to these systems. Rather, system specific tools must be developed to fuzz test and observe the status of these systems.

Secondly, these systems employ a large variety of complex, often proprietary, protocols that are not found in conventional computer networks. Since these protocols are less prevalent, existing fuzzing tools do not support them. These protocols have less exposure in academia and to security researchers more broadly so vulnerability testing is performed on them far less often.

In light of these challenges, the report highlights two fuzzing frameworks that are widely used and may be adaptable to embedded system fuzzing. The preeminence of these two frameworks is confirmed by their mention elsewhere in the literature, including in the three aforementioned modern fuzzing applications, [9], [12], and [16], and cyber-security researchers awareness of them in general. The two tools are Sulley, [7], and Peach, [10].

2.1.2.1 Sulley.

Sulley is an open source, Python based, fuzzing framework that consists of many components [7]. The framework provides tools to generate fuzzed messages, deliver them, and monitor the network and receiving applications for adverse effects. To generate fuzzed data, Sulley takes a block based approach like that of the original open source fuzzer, SPIKE [8]. The user is able to model a protocol using simple or complex field types that can then be individually mutated. The block based approach also allows the user to model dependencies between fields like checksums.

Sulley natively supports delivering fuzzed messages via TCP, UDP and SSL. Other

delivery mechanisms are possible but require the user to program a python module. State modeling of the network protocols is also supported allowing the fuzzer to handle more complex protocol interactions. Sulley places a large emphasis on monitoring capabilities. These tend to focus on packet capture and analysis as well as application fault detection tools.

The block based method for protocol modeling may be adaptable to military data link protocols. If such a method was implemented, Sulley could be used to automatically generate test cases based on a model. However, Sulley may not be very helpful from that point onward. Since many data link protocols do not use TCP or UDP, travel over common data link layer standards like Ethernet or IEEE 802.11, or are even packet based; the delivery mechanisms native to Sulley are not of much use. Similarly, Sulley's monitoring capabilities focus on traditional packet switched computer networks and would not be useful in the embedded systems like military data links. For these reasons, Sulley was not pursued as a fuzzing framework for this thesis.

2.1.2.2 Peach.

Peach is a fuzzing framework that has been in development since 2004 [10]. An open source community version exists as well as a commercial version. Peach takes a different approach to protocol modeling than Sulley. Its approach involves the development of an XML document called a Peach Pit. A Peach Pit contains information about the protocol to be fuzzed including information about the data fields as well as state information for more dynamic protocols.

Peach contains a larger set of built in delivery mechanisms than Sulley, and custom mechanisms can be written if necessary. Peach also has some monitoring capabilities, called agents, that can attach debuggers to a target process or monitor for error

messages or other signs that a target has been affected.

Like Sulley, the delivery and monitoring features of Peach are not particularly applicable to military data link fuzzing, but modeling these protocols with Peach Pits is feasible. The data model inside a Peach Pit is designed to be human readable and is implemented in a flexible XML structure. For this reason, the Peach Pit was chosen as a starting point for the protocol models developed in this thesis. More discussion on this is found in Section 4.1.1. Unfortunately the fuzzing functions, called mutators, that Peach applies to the data model to generate fuzz test sets are all designed with traditional computer network protocols in mind. The Peach framework does allow for the development of custom mutators; such a step would be necessary before using Peach in a military data link fuzz test.

2.2 Adequacy Criterion

2.2.1 In Software Testing.

Software testing has been around as long as software has. As early as the 1970s, researchers tried to identify the best methods for selecting test cases to test programs for errors and bugs. The fundamental question arrived at by these efforts was "What is the test criterion", that is, what is measured to determine the adequacy of a test [11]. In the 1990s a survey paper, [29], identified various test criterion that have been proposed and how they can be used to demonstrate test adequacy. This paper further uses the term coverage with regards to the idea of test adequacy. For example, if the test adequacy criterion is to exercise every statement in a program, this criterion is called the statement coverage criterion. A percentage can be used to describe how many of the statements have been exercised in a test to indicate how adequately testing has been performed.

2.2.2 In Fuzzing.

Software testing criterion tend to focus on white-box testing, that is using information about the source code of the software under test to develop test cases and measure coverage [19]. There are not many definitions of adequacy criterion for the black box case. It seems in most cases when black box fuzzing methods are developed, their adequacy criterion is not defined, and the measure of effectiveness of their methods is based on how many new vulnerabilities they were able to find as in [17], [15] and [25], or the fact that they can generate any possible input even though generating every input would be infeasible as in [16].

In a couple of cases, the adequacy criterion and coverage measures were defined for black box fuzzing. First, [14] represents network protocol specifications as finite state machines and has an adequacy criterion of all transitions fuzzed. It also defines the adequacy criterion of all message types fuzzed, but claims this is inferior to the transition criterion because it ignores the state machine nature of the protocol. This criterion hinges on the system under test being easily and sufficiently modelled using a finite state machine which may be difficult due to the black box nature of the system.

Second, [26] defines semi-valid input coverage (SVCov) for fuzz testing and claims that it is the first coverage criterion for fuzz testing. The core idea behind SVCov is that the protocol specification can be written as a finite set of constraints on the input messages. Since effective fuzzing requires inputs that at least vaguely resemble valid messages so that they do not get immediately dropped by the receiving system, SVCov only measures how many test cases break one and only one of the constraints on the input set. This is why the method is named semi-valid input coverage.

This approach has many strengths. First of all, the method is straight forward; once a set of constraints is derived from a protocol specification, the coverage computation is simply the number of constraints broken by the test set divided by the total

number of constraints. Second, the method is generic enough to cover most types of fuzzing including file fuzzing, software input fuzzing, as well as protocol fuzzing. Finally, SVCov operates only a set of test cases; it is not tied to the method used to generate them. This allows it to fit in the fuzzing framework described in Section 1.4. The method takes two inputs: a protocol specification (used to derive the constraints) and the set of test cases, and returns one output: the coverage value, a percentage between 0% and 100%.

Despite these strengths, this coverage criterion has a few faults that may preclude it from being considered a viable measure of test set effectiveness. First, the authors acknowledge that the creation of a non-redundant set of constraints may be challenging for complex protocols. Since the method does not consider any test cases that break two or more constraints, if two constraints are redundant so that neither can be broken while the other is not, these constraints will never be covered. Data link protocols are complex, and it is probably a non-trivial effort to derive non-redundant constraints.

Second, the relation between the coverage value and the proportion of existing vulnerabilities the test set should uncover is hard to discern. The tester would know that a test set with an SVCov score of 90% can be expected to uncover more vulnerabilities than a test set with a score of 60%. But counter-intuitively, the test set scoring 90% by no means has tested 90% of the invalid inputs, nor is it expected to find 90% of the vulnerabilities, nor has it a 90% chance of finding a vulnerability. The 90% figure is instead based on an abstract representation of the protocol as constraints. It may be hard to base further testing decisions on this figure.

Third, the authors also acknowledge that 100% coverage based on SVCov is in no way a guarantee that all existing vulnerabilities existing in the protocol have been exposed. Consider these two examples of invalid inputs that may trigger a

vulnerability, but would not be covered by the SVCov metric.

1. A protocol has a constraint that says a message may be no more than 3 words long and another constraint that says a command word may not appear more than twice in a message. A vulnerability is triggered when four or more command words appear in the same message. Any test case that would discover this vulnerability would break both constraints and therefore provide no coverage under SVCov.

2. A protocol has a "second" field in which the second of the current minute is reported. A constraint that says the value in this field may not exceed 59. Another field rates the confidence in the value being reported in the first field. Unbeknown to the tester that derived the constraints, the SUT only processes the "second" field if the confidence level in the other field is high enough. A test case that breaks the constraint and sends a second value of 62, but includes a low confidence value, is not processed by the SUT, but is marked as covering the broken constraint. Thus, a test set may break every constraint, but leave a vulnerability undiscovered.

As the authors suggest, SVCov is best when paired with other coverage criterion to accurately assess the effectiveness of a fuzz test. Unfortunately very few such criterion exist. In [26], the authors go on to suggest a number of non-fuzzing specific criterion that may apply, but these tend to focus on white box testing methods that require access to source code.

2.2.3 Fundamental Axioms.

Eight fundamental axioms of software testing adequacy are laid out by [27] and can be used as a basis for building a coverage criterion. Axioms one, three and four

from this paper are applicable to fuzz testing (the other five are not) and [26] adds a another axiom specific to fuzz testing. Together they are:

1. Applicability: There is at least one finite set of test cases that is adequate, that is it provides 100% coverage.
2. Monotonicity: If a test set is enlarged its coverage may not decrease.
3. Inadequacy of the Empty Set: The empty test set provides zero coverage.
4. Fuzz-test coverage: A test set consisting of only valid inputs achieves zero coverage

Any fuzzing coverage criterion must meet these fundamental axioms. This thesis presents a new approach that satisfies these fundamental axioms and provides a coverage criterion designed to be an accurate reflection of the thoroughness of a test set. Unlike the approach taken in [14], this approach does not use a finite state model. Doing so would increase the robustness of the analysis, but would be difficult to achieve for the black-box case where state information is often unattainable. This new approach does not require any constraints to be manually defined as in [26] except a delineation between the valid and invalid inputs for each field.

III. Creating the Criterion

In this chapter a coverage criterion for fuzz testing called ExCov is presented. This thesis defines coverage as *the expected percent of existing vulnerabilities discovered by a set of test cases*. To develop the criterion, this definition is unpacked and implemented. In this chapter, a procedure is created that takes as inputs a set of test cases and a data model and returns a value between 0% and 100%. This value is the best estimate of the expected percentage of existing vulnerabilities that test set would discover when applied to a SUT using the protocol in question. The equations and procedures presented in this chapter are implemented in a comprehensive example in Appendix C.

The approach taken in this chapter treats a protocol as a set of fields. A vulnerability in the protocol is assumed to manifest in either one field, or a combination of fields. For example, if a SUT has a vulnerability where a value of 0xFFFF in field 6 triggers an error state; that vulnerability would be said to reside in field 6. If an error state is only entered if field 5 has a value of 0x42 AND field 6 has a value of 0xFFFF, then the vulnerability is said to reside in the field combination of fields 5 and 6.

This difference between fields and field combinations will be covered extensively later in the chapter, but it is important to understand how vulnerabilities are classified so the approach taken to compute coverage is clear. The ExCov procedure starts with the idea of field coverage. That is, the coverage of each individual field can be calculated in isolation. By limiting the scope of the coverage calculation to a single field, the necessary equations and procedures to compute coverage are easier to derive and explain. This is done in Section 3.1. Once the procedures and equations for computing the coverage of each field are presented, the coverage values must be combined to arrive at the coverage for all fields. This is done in Section 3.2.1.

Vulnerabilities are not necessarily restricted to only one field however. As in the

previous example, some vulnerabilities may span multiple fields. These vulnerabilities are called multi-order vulnerabilities. More specifically, an N^{th} order vulnerability spans N fields. To handle these vulnerabilities, a procedure was created to measure coverage for field combinations. Notationally, a field combination of N fields is said to be an N^{th} order field combination. Coverage of multi-order field combinations is presented in Section 3.2.2, with special cases handled in Section 3.2.3. The coverage values for fields and field combinations must be combined to reach a total coverage value for the test set. This is done in Section 3.2.4.

An underpinning mechanic of this coverage calculation method is the modeling of vulnerabilities in general. An approach to this task is shown in Section 3.3. Finally, Section 3.4 synthesizes the information from the rest of the chapter into a step by step procedure for calculating coverage.

3.1 Field Coverage

This thesis defines field coverage as *the expected percentage of existing vulnerabilities in a field discovered by a set of test cases.* This definition needs to be unpacked.

First, the meaning of the term *discovers* needs to be clarified. A test case is said to *discover* some vulnerability V if it is the first test case in a test set to test a value that triggers V. For example, consider a field where a vulnerability V is triggered if a 5, 6, or 7 is tested. A test set of three test cases is applied. The first test case tests a value of 4, the second test case tests a value of 7, and the third test case tests a value of 6. Test case 1 does not discover V because it does not test 5, 6, or 7. Test case 3 does not discover V because V was already discovered when the test case was applied. Test case 2 does discover V since it was the first test case to test 5, 6, or 7. A vulnerability is *discovered by a set of test cases* if it is discovered by any test case in the set of test cases.

Second, the term *expected* refers to the concept of probabilistic expectation. This implies that there is some sort of randomness or unknown parameter in the fuzz test over which an expectation can be found. Consider the contrary case of an omniscient coverage calculator, that is, a calculator that knows all test case inputs and all vulnerabilities in the protocol. In this case, the number of vulnerabilities a set of test cases discovers can be found directly, and therefore field coverage definition no longer requires the *expected* qualifier. *The percentage of existing vulnerabilities in a field discovered by a set of test cases* can be directly computed.

In Section 3.1.1 a field is considered where the vulnerabilities are known, but the test cases are randomly selected. In this case, field coverage can be accurately calculated with a random model for test case selection. Derived from this example are the equations necessary to calculate field coverage for any number of random test cases in a field with known vulnerabilities.

Of course, if the vulnerabilities are known, fuzz testing is unnecessary. Further, this set-up for a coverage calculator is not the one presented in the fuzzing framework in Section 1.4. In that framework, the test cases are known to the calculator, but the vulnerabilities in a field are not. To account for these unknown vulnerabilities, a probabilistic model for vulnerabilities in general is created in Section 3.3.

To find the equations that measure field coverage under this framework, the equations that compute field coverage in Section 3.1.1 are modified to shift from expectation over random test cases to expectation over the random model for vulnerabilities. This is done in Section 3.1.2. The equations arrived at in Section 3.1.2 are sometimes not directly applicable to special types of fields or fields where the tester has some information about possible vulnerabilities. Three of these cases that arose during the course of this thesis are described and accounted for in Section 3.1.3.

3.1.1 Coverage With Known Vulnerabilities.

This section considers a made-up field where the vulnerabilities are known, but the test cases are randomly selected. The primary purpose of this section is to derive basic equations for field coverage, but it also serves as an introduction to many of the terms created for this thesis and used throughout this chapter to describe the abstract concepts behind crafting a fuzz testing criterion. A list of these terms and their definitions can be found in Appendix A.

The field used in this section and its associated spaces are shown in Figure 4. The field contains 5 bits, so there are 2^5 or 32 possible values this field can take. The protocol standard for this field does not allow two ones to be adjacent to each other. All values that satisfy the standard are considered elements in the field's *valid space*. If a value does not satisfy the standard, and is therefore not in the valid space, it is in the *invalid space*

Notationally, any space associated with a field is a set with elements that are the values the field can take on. In this field, the invalid space is set I where:

$$I = \{00011, 00110, 00111, 01011, 01100, 01101, 01110, 01111, 10011, 10110,$$
$$10111, 11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111\}$$

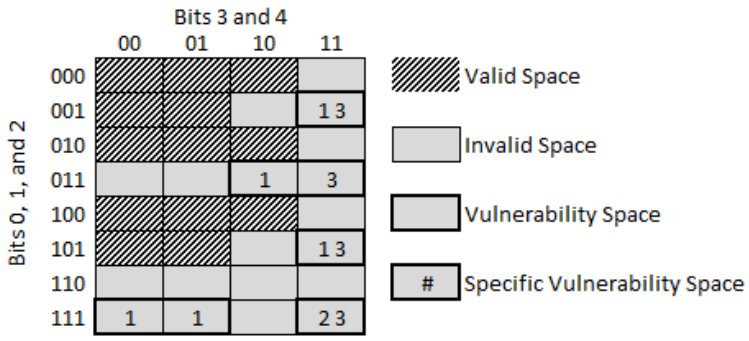

Figure 4. An illustration of the spaces associated with a five bit field

Usually in this thesis field values will be represented in decimal, not binary. This way I can be more clearly defined as:

$$I = \{3, 6, 7, 11, 12, 13, 14, 15, 19, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31\}$$

It is much more common in this chapter for the cardinality of I, denoted $|I|$ to be used than I alone. This simply means the number of elements in I. For this field $|I| = 19$.

The invalid space of a field may contain zero, one or many vulnerabilities. In this example, the vulnerabilities present in this field are known. Three vulnerabilities can be triggered by values in this field:

Vulnerability 1: There are exactly three adjacent ones.

Vulnerability 2: All bits are ones.

Vulnerability 3: The last three bits are ones.

The set of values that trigger a specific vulnerability make up what is called the *specific vulnerability space* of that vulnerability. For example, the specific vulnerability space of vulnerability 1, call it V_1 is:

$$V_1 = \{00111, 01110, 10111, 11100, 11101\} = \{7, 14, 23, 28, 29\}$$

The cardinality of these spaces is also important. For this vulnerability, $|V_1| = 5$. The cardinality of a space will also be called the size of a space in this thesis. Note that specific vulnerability spaces are not necessarily mutually exclusive. The value 7 is in both V_1 and V_3.

The *vulnerability space* is the set of all values that trigger a vulnerability. In general, a vulnerability space is the union of all specific vulnerability spaces. For this

field, vulnerability space V, is:

$$V = V_1 \cup V_2 \cup V_3 = \{7, 14, 15, 23, 28, 29, 31\}$$

In this section, field coverage for this field will be calculated for test sets with 1 to 19 randomly selected test cases. The values for the test cases will always be in the invalid space of the field, with each invalid value having an equal probability of selection. Since field coverage is defined as the expected percentage of vulnerabilities discovered, probabilistic expectation will be incorporated into the calculation. This will allow the randomness of the inputs to be handled theoretically and remove the need for any simulation.

The calculation will begin by finding the coverage of only one random test case in Section 3.1.1.1, then it will extend this to a test set of two random test cases in Section 3.1.1.2. Finally the calculation will be extended to incorporate any number of test cases in Section 3.1.1.3.

3.1.1.1 One Test Case.

Before starting the field coverage calculation some variables need be defined:

K_i: the number of vulnerabilities discovered by test case i

N: the number of vulnerabilities present in the field

P_j^i: the probability that test case i discovers exactly j vulnerabilities

From these definitions, the expected number of vulnerabilities discovered by test case 1 can be simply denoted as $E[K_1]$. Using the definition of expected value, the expected number of vulnerabilities discovered by test case 1 can be calculated as:

$$E[K_1] = \sum_{n=0}^{N} nP_n^1 \tag{1}$$

By examining Figure 4, the number of vulnerabilities each value in the invalid space will discover can be easily found. Since each value is equally likely to be tested, the probability the first test case will discover 0, 1, and 2 vulnerabilities can be found. This is all shown in Table 1.

Table 1. Possible outcomes from the first test case broken down by number of vulnerabilities discovered

Number of vulnerabilities discovered	0	1	2
Test cases that discover them	11000,11001,00011, 01011,11011,10011, 11010,00110,11110, 10110,01101,01100	01110,01111,11101, 11100	00111,11111,10111
Probability of discovering that many vulnerabilities	$P_0^1 = {}^{12}/_{19}$	$P_1^1 = {}^{4}/_{19}$	$P_2^1 = {}^{3}/_{19}$

Applying (1) to the example, the expected number of vulnerabilities discovered by one test case can be found.

$$E[K_1] = \sum_{n=0}^{N} nP_n^1 = 0\,({}^{12}/_{19}) + 1\,({}^{4}/_{19}) + 2\,({}^{3}/_{19}) + 3\,(0) = {}^{10}/_{19} \tag{2}$$

From the definition of field coverage, the coverage of the first test case, call this C_1, is the expected percentage of vulnerabilities found by the first test case. Therefore by dividing the number of discovered vulnerabilities, K_1 by the total number of vulnerabilities, N, the field coverage for one test case is found.

$$C_1 = E\left[\frac{K_1}{N}\right] = \frac{E[K_1]}{N} = \frac{{}^{10}/_{19}}{3} = 17.5\% \tag{3}$$

There is another way to calculate coverage that is less straightforward but is easier to implement than the previous approach. It involves looking at which vulnerabilities a test case will uncover instead of how many. First, consider the expression $Pr(T_i \subseteq \tilde{V_j})$ where T_i is a set of one value, test case i, and $\tilde{V_j}$ is undiscovered vul-

nerability j. This reads as the probability that the value of test case i is in the specific vulnerability space of the so far undiscovered vulnerability j. Said more simply, the probability test case i discovers V_j.

The probability the first test case discovers vulnerability j is simply:

$$Pr(T_1 \subseteq \tilde{V_j}) = \frac{|V_j|}{|I|} \tag{4}$$

Using (4) and Figure 4, the probability each vulnerability is discovered for the field is:

$$Pr(T_1 \subseteq \tilde{V_1}) = 5/19 \quad Pr(T_1 \subseteq \tilde{V_2}) = 1/19 \quad Pr(T_1 \subseteq \tilde{V_3}) = 4/19 \tag{5}$$

Notice that the sum of these probabilities is also the expected number of vulnerabilities found:

$$\sum_{n=1}^{N} Pr(T_1 \subseteq \tilde{V_n}) = 5/19 + 1/19 + 4/19 = 10/19 = E[K_1] \tag{6}$$

The proof in Figure 5 shows that this new way to calculate the expected number of vulnerabilities found in (6) is equivalent to the more straightforward way in (2). That is:

$$E[K_i] = \sum_{n=0}^{N} nP_n^i = \sum_{n=1}^{N} Pr(T_i \subseteq \tilde{V_n}) \tag{7}$$

3.1.1.2 Two Test Cases.

The coverage of the field for two test cases is a bit trickier to compute. First it must realized that $Pr(T_2 \subseteq \tilde{V_j})$ depends on $Pr(T_1 \subseteq \tilde{V_j})$ because if test case 1 discovers a vulnerability, test case 2 can't discover that vulnerability. Therefore $Pr(T_2 \subseteq \tilde{V_j})$ becomes:

Let I be the set of all distinct values in the invalid space, and let T_i be a set of cardinality 1 where $T_i \subseteq I$ and T_i represtents the value tested by test case i.
Let $\tilde{V}_1, \tilde{V}_2, \ldots, \tilde{V}_N$ be subsets of I, the specific vulnerability spaces that have yet to be discovered.
Let $\tilde{V}_j = \emptyset$ if $T_k \subseteq \tilde{V}_j$ for any $1 < k < i$, and any $1 < j < N$. That is, if vulnerability j has been discovered by a previous test case, $\tilde{V}_j$ is the empty set since $\tilde{V}_j$ represents specific vulnerability spaces not yet discovered.
Let the cardinality of the intersection between T_i and $\tilde{V}_j$ be denoted $X_{i,j}$:

$$X_{i,j} = \left| T_i \cap \tilde{V}_j \right|$$

That is, $X_{i,j} = 1$ if test case i discovers vulnerability j, and $X_{i,j} = 0$ otherwise. From this definition it follows that:

$$Pr\left(X_{i,j} = 1\right) = Pr(T_i \subseteq \tilde{V}_j)$$

It also follows that the number of vulnerabilities discovered by test case i, K_i, can be represented as:

$$K_i = \sum_{n=1}^{N} X_{i,n}$$

The expected number of vulnerabilities discovered by test case i is therefore:

$$E\left[K_i\right] = E\left[\sum_{n=1}^{N} X_{i,n}\right]$$

By the linearity of the expectation operator:

$$E\left[\sum_{n=1}^{N} X_{i,n}\right] = \sum_{n=1}^{N} E\left[X_{i,n}\right]$$

By the definition of expected value:

$$E\left[X_{i,n}\right] \;=\; \sum_{m=0}^{1} m \cdot Pr\left(X_{i,n} = m\right) \;=\; 1 \cdot Pr\left(X_{i,n} = 1\right) \;=\; Pr(T_i \subseteq \tilde{V}_n)$$

Therefore:

$$E\left[K_i\right] = \sum_{n=1}^{N} Pr(T_i \subseteq \tilde{V}_n)$$

Figure 5. Proof for the alternate definition of expected number of vulnerabilities found

$$Pr(T_2 \subseteq \tilde{V_j}) = Pr\left(Discovering\ V_j\ \cap\ V_j\ not\ discovered\ yet\right) \tag{8}$$
$$= Pr\left(Discovering\ V_j \mid V_j\ not\ discovered\ yet\ \right) Pr\left(V_j\ not\ discovered\ yet\right)$$

$Pr\left(Discovering\ V_j \mid V_j\ not\ discovered\ yet\ \right)$ is the probability test case 2 discovers V_j, when test case 1 did not. This is simply the size of the vulnerability, $|V_j|$ divided by one less than the size of the invalid space, $|I|$, since there are one less values to choose from for the second test case. $Pr\left(V_j\ not\ discovered\ yet\right)$ is simply $1 - Pr\left(V_j\ discovered\ already\right) = 1 - Pr(T_1 \subseteq \tilde{V_j})$

So in general for test case 2:

$$Pr(T_2 \subseteq \tilde{V_j}) = \frac{|V_j|}{|I| - 1}\left(1 - Pr(T_1 \subseteq \tilde{V_j})\right) \tag{9}$$

For the example:

$$Pr(T_2 \subseteq \tilde{V_1}) = \frac{5}{18}\left(1 - \frac{5}{19}\right) = 0.20 \tag{10}$$
$$Pr(T_2 \subseteq \tilde{V_2}) = \frac{1}{18}\left(1 - \frac{1}{19}\right) = 0.05 \tag{11}$$
$$Pr(T_2 \subseteq \tilde{V_3}) = \frac{4}{18}\left(1 - \frac{4}{19}\right) = 0.18 \tag{12}$$

Using (7):

$$E\left[K_2\right] = \sum_{n=1}^{N} Pr(T_2 \subseteq \tilde{V_n}) = 0.20 + 0.05 + 0.18 = 0.43 \tag{13}$$

Coverage of a field with two test cases is the expected number of vulnerabilities discovered by either test case divided by the total number of vulnerabilities, therefore:

$$C_2 = \frac{E[K_1] + E[K_2]}{N} = 32.0\% \tag{14}$$

3.1.1.3 M Test Cases.

Using the same logic as above, these equations can be extended to any number of test cases:

$$Pr(T_i \subseteq \tilde{V_j}) = \begin{cases} \frac{|V_j|}{|I|}, & i = 1 \\ \frac{|V_j|}{|I|-i+1}\left(1 - \sum_{k=1}^{i-1} Pr(T_k \subseteq \tilde{V_j})\right), & i > 1 \end{cases} \tag{15}$$

$$C_m = \frac{1}{N}\sum_{i=1}^{m} E[K_i] = \frac{1}{N}\sum_{i=1}^{m}\sum_{n=1}^{N} Pr(T_i \subseteq \tilde{V_n}) \tag{16}$$

Figure 6 shows the application of (15) and (16) to the field. The figure plots the coverage of $C_1, C_2, \ldots, C_{|I|}$, or test sets of size 1, 2, ..., 19. As expected, coverage only increases as more test cases are added. Since later test cases have a higher chance of having values in the specific vulnerability spaces of already discovered vulnerabilities, they have less of a chance of discovering new vulnerabilities. This means that they provide less marginal coverage. This can be seen in the figure as each successive test set size produces smaller increases in coverage.

3.1.2 Estimating Coverage.

The field coverage equations (15) and (16) are of little direct use. Testers will not have access to the underlying vulnerability space; if they did fuzz testing would be unnecessary. Testers will not have any way of finding $|V_j|$ or N, so (15) and (16) can not be computed.

However by making some observations and assumptions about the nature of vulnerability spaces and, by extension, the values of $|V_j|$ and N an estimate of coverage

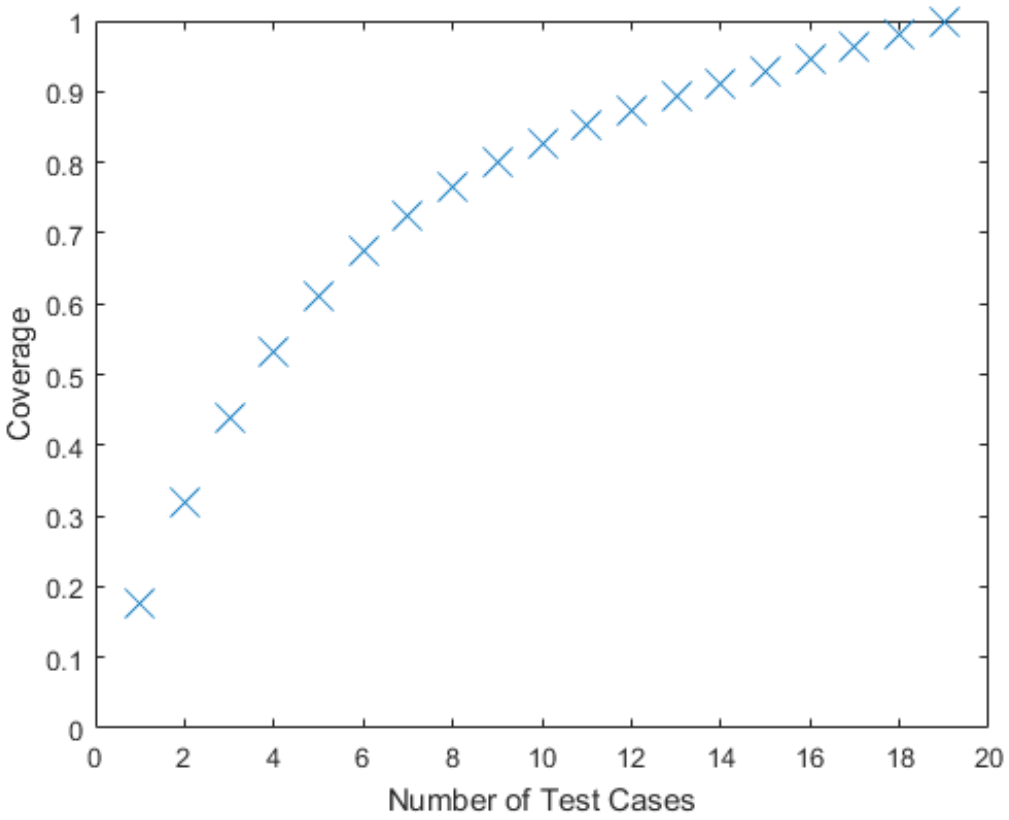

Figure 6. Coverage of the five bit example field

can be found. Research into real vulnerabilities in protocols can reveal information about the general nature of vulnerability spaces and help focus these assumptions, but this information is difficult to quickly aggregate. Working with the information available, a best guess can be made for the underlying distribution of vulnerabilities, and a coverage value can be computed from these estimates.

Looking at (15) and (16) it can be seen that the size of each vulnerability, $|V_j|$, and the number of vulnerabilities, N, are the two variables that contain information about the vulnerability space. Therefore, these are the two variables that a probabilistic estimate of vulnerabilities must account for.

Further analysis reveals that coverage, C_m, depends on the distribution of vulnerability sizes alone and not the number of vulnerabilities, in other words, C_m depends

on $|V_j|$ but not directly on N. For example, fields where 50% of the vulnerabilities are of size S_1 and 50% of the vulnerabilities are of size S_2 both tested with m test cases, will always have the same coverage regardless of N.

This is shown in the following equations. First consider that, based on (15), $Pr(T_i \subseteq \tilde{V_j})$ depends on the size of the vulnerability, $|V_j|$, but nothing else about V_j. Therefore:

$$\text{If } |V_a| = |V_b|, \text{ then } Pr(T_i \subseteq \tilde{V_a}) = Pr(T_i \subseteq \tilde{V_b}) \tag{17}$$

Next, let half of the vulnerabilities be of size S_1, call them V_{odd}, so that $|V_{odd}| = S_1$. Let the rest be of size S_2, call them V_{even}, so that $|V_{even}| = S_2$.. Then (16) can be simplified to remove its dependence on N.

$$C_m = \frac{1}{N} \sum_{i=1}^{m} \left(\sum_{j \ is \ odd} Pr(T_i \subseteq \tilde{V_j}) + \sum_{j \ is \ even} Pr(T_i \subseteq \tilde{V_j}) \right) \tag{18}$$

$$C_m = \frac{1}{N} \sum_{i=1}^{m} \left(\frac{N}{2} Pr(T_i \subseteq \tilde{V_1}) + \frac{N}{2} Pr(T_i \subseteq \tilde{V_2}) \right) \tag{19}$$

$$C_m = \sum_{i=1}^{m} \left(\frac{1}{2} Pr(T_i \subseteq \tilde{V_1}) + \frac{1}{2} Pr(T_i \subseteq \tilde{V_2}) \right) \tag{20}$$

Coverage therefore depends on the distribution of vulnerability sizes and not characteristics of the individual vulnerabilities or the number of total vulnerabilities. A probabilistic model of these vulnerability sizes can be built that allows coverage to be calculated based on what is known about vulnerabilities in general instead of the specific vulnerabilities of the field in question.

That model can come in the form of a Probability Mass Function (PMF) which describes the probability a vulnerability has a certain size. Formally, let X be a random variable that maps specific vulnerability spaces to their cardinality, $|V_j|$. Due

to the definition of specific vulnerability spaces, X must take on a value between 1 and $|I|$. The distribution of this PMF is determined by the nature of all vulnerabilities as well as any specific information the tester has about the fields in question.

Using this PMF the approach taken to find equations (15) and (16) can be re-assessed. Instead of random test cases with known vulnerabilities, lets assume that each vulnerability size, $|V_j|$ is a random draw from the distribution described by the PMF. Then the probability test case i discovers vulnerability j becomes:

$$\begin{aligned} Pr(T_i \subseteq \tilde{V_j}) &= \sum_{x=1}^{|I|} Pr(T_i \subseteq \tilde{V_j} \text{ AND } x = |V_j|) && (21) \\ &= \sum_{x=1}^{|I|} Pr(|V_j| = x) Pr(T_i \subseteq \tilde{V_j} \mid x = |V_j|) && (22) \\ &= \sum_{x=1}^{|I|} f_X(x) Pr(T_i \subseteq \tilde{V_j} \mid x = |V_j|) && (23) \end{aligned}$$

$Pr(T_i \subseteq \tilde{V_j} \mid x = |V_j|)$ is simply (15) with x replacing every $|V_j|$. It can be rewritten as a new function $R(x,i)$ that is a substitution into (15). This function is shown in (24).

$$R(x,i) = \begin{cases} \frac{x}{|I|} & i = 1 \\ \frac{x}{|I|-i+1}\left(1 - \sum_{k=1}^{i-1} R(x,k)\right) & i > 1 \end{cases} \quad (24)$$

Using this function in (23):

$$Pr(T_i \subseteq \tilde{V_j}) = \sum_{x=1}^{|I|} f_X(x) R(x,i) \quad (25)$$

Once this is substituted into (16), there is no longer any dependence on the index of the vulnerability (j in (15) and n in (16)). This makes sense because the vulnerabilities are modeled using the PMF instead of finding coverage for N specific known

vulnerabilities. This allows for the following simplification:

$$C_m = \frac{1}{N}\sum_{i=1}^{m}\sum_{n=1}^{N}\sum_{x=1}^{|I|} f_X(x)R(x,i) \tag{26}$$

$$C_m = \frac{1}{N}\sum_{i=1}^{m} N\sum_{x=1}^{|I|} f_X(x)R(x,i) \tag{27}$$

$$C_m = \sum_{i=1}^{m}\sum_{x=1}^{|I|} f_X(x)R(x,i) \tag{28}$$

With the dependence on N removed, there is no longer any dependence on unknowable values. This allows the field coverage calculation shown in (29) to be applied to any field that does not require special modifications discussed later in this chapter.

$$C_m = \sum_{i=1}^{m}\sum_{x=1}^{|I|} f_X(x)R(x,i) \tag{29}$$

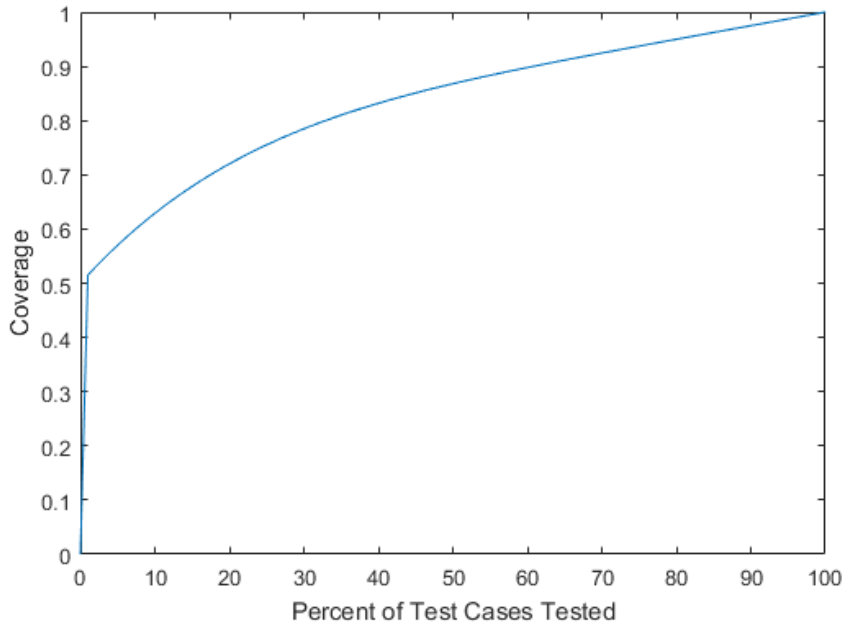

Figure 7. Coverage of a field with $|I| = 100$ for all possible test set sizes using the vulnerability model described by (30)

As an example to illustrate how to compute field coverage using a model for vulnerability sizes, consider a field which has an invalid space with $|I| = 100$. Nothing about the vulnerabilities present in this specific field are known, but a model for vulnerabilities in general says, simply, 25% of vulnerabilities have a size of 1, 25% have a size of 5, and 50% cover the entire invalid space. This model for vulnerability sizes would look like:

$$f_X(x) = 0.25\delta(x-1) + 0.25\delta(x-5) + 0.5\delta(x-|I|) \tag{30}$$

where $\delta(x)$ is the discrete unit impulse function. When applied to the estimated coverage equations, (24) and (29), the curve in Figure 7 is produced.

A much better model for vulnerability sizes is described in Section 3.3. Using that model, and equations (24) and (29), field coverage for a set of test cases can be calculated for fields that are not considered special cases.

3.1.3 Special Cases.

3.1.3.1 Structure Fields.

Fields that contain information about how a message is structured or interpreted are defined as structure fields in this thesis. Sometimes these fields have valid and invalid spaces that can not be simply defined based on valid and invalid values. For example, consider the 3 bit "Number of Words" field in the restaurant locator protocol described in Table 12 on page 151. This field is a structure field because it is related to the number of blocks of fields (called words for this protocol) to follow. A value greater than five in this field is always invalid because there are only five words that are allowed to follow the first word. A value of say, two, is only valid if two words follow the first word. So if in one message the field has a value of two, but three

words follow, the field is invalid. But the next message could have the same value of two, with two words following and therefore be valid.

Clearly the field coverage calculation for basic fields, (24) and (29), cannot be applied if values are valid or invalid depending on more than just the value itself. For these fields, valid and invalid spaces will be computed differently. With an accurate definition of these spaces, the same approach used previously to compute the coverage of a basic field can be applied to a structure field as well.

For this thesis only one type of structure field, a repeating relation field, required a special invalid space definition. Other structure fields can be defined using the data model described in Chapter IV that would require a unique definition of their invalid space, but due to time restraints and a lack of a direct need, these definitions were not developed. Some examples of such a structure field types would be the total length and checksum fields of the IPv4 protocol.

A repeating relation field is a field whose value dictates the number of times a block of fields repeats. The "Number of Words" field in the Restaurant Locator Protocol is an example of such a field. This type of field is defined in a data model by using the "Relation" element with three attributes:

1. **type:** the type of relation; for a repeating relation field this is "count", a reference to the count of the number of times the linked block repeats. This attribute comes from the Peach language.

2. **of:** the block that is repeating, also from the peach language [10].

3. **adjustment:** the difference between the number of times the related block repeats and the value in the field. This attribute was added to handle cases where this value is not zero.

The block that repeats has two attributes that are also necessary for determining

the repeating relation field's valid and invalid spaces. They are the "minOccur" and "maxOccur" attributes, and describe the number of possible times a block can repeat. It is important to note that this is not the mininum and maximum number of times it is allowed to repeat per the standard, but the minimum and maximum times it can repeat given testing, hardware and SUT parsing limitations. For example, the Restaurant Locator protocol has a minOccurs of 1 and a maxOccurs of 8 for its Outer Repeating Block. Any more than 6 repeats would be invalid, but it is known or suspected that the SUT may accept as many words as any value in the "Number of Words" field would allow. Perhaps the standard was written to allow for more words to be added later as needed, and the SUT will try to process up to 8 words.

With knowledge of the relation's set-up, the valid and invalid spaces can be created. For repeating relation fields, a test case where the field has a value of two, and the related block repeats 3 times, is distinctly different from the test case where the field has a value of two and the related block repeats 4 times. Thus these two cases are separate elements in the field's valid or invalid space. Another possibility is that, due to a choice somewhere in the data model, the related field is missing from the test case. This type of occurrence is definitely invalid, and it is considered a unique element of the invalid space for every possible number of times the related block can repeat. Figure 8 shows the adjusted valid and invalid spaces for the repeating relation field, Number of Words in the Restaurant Locator Protocol.

3.1.3.2 Numeric Fields.

Another unique type of field that is worth re-evaluating is a numeric field. In this type of field, values have a numerical meaning defined by the protocol standard. For example, consider a field in a protocol that represents the number of years since 2000 when the message is being sent. The field is 8 bits long so theoretically any year from

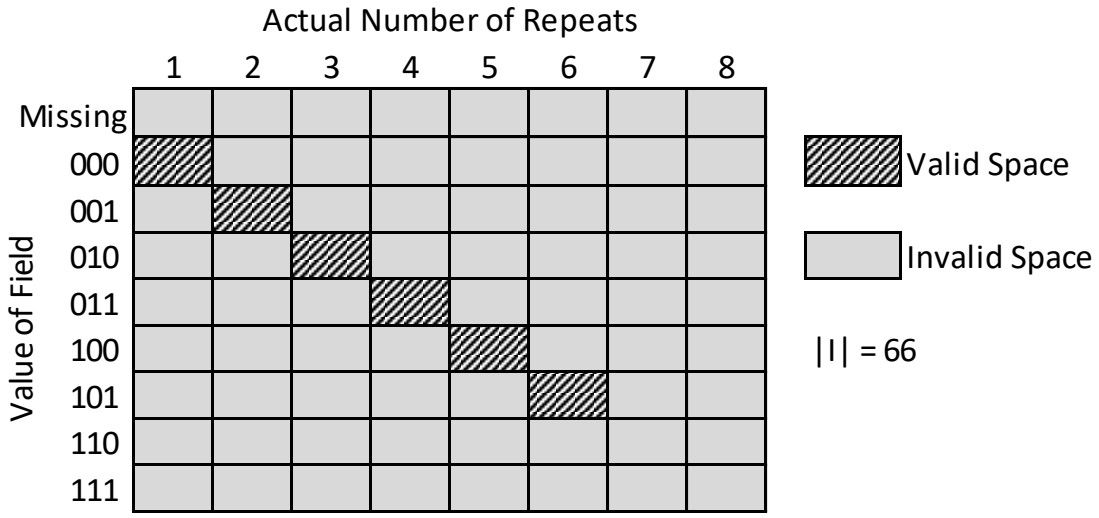

Figure 8. The spaces associated with the Number of Words field from the Restaurant Locator Protocol

2000 to 2255 can be represented. The system is designed to last until 2075, so a value in this field above 75 is unexpected and therefore invalid. It could be imagined that the system might combine this information with a day and a month from some other fields and compute a day of the week. If the program that does this calculation uses a lookup table and that table only goes to the year 2100, any number over 100 may cause an issue in this system. A vulnerability that exploits this issue would have a specific vulnerability space of all values between 101 and 255.

In fields like this, test cases can no longer be considered equally likely to discover a vulnerability. In this example, some test cases that are larger, and further from the valid space are more likely to discover a vulnerability. This violates the assumption previously made that two different invalid inputs are equally likely to trigger a vulnerability. If it is known that a field is numeric, then some test cases are more likely to discover vulnerabilities than others.

More extreme values are not necessarily more likely to uncover vulnerabilities however. Consider the system previously described, and suppose the program computing day of the week was well written to ignore any input with a year beyond 2100.

However, this program sends the full date, with the year and day of the week to another program that acts on it. If this second program is designed to only expect years equal to or less than 2075, and it receives a date with the year 2090, it may react poorly, revealing a vulnerability. In this case a vulnerability exists with a space containing all values between 76 and 100.

These two examples shows another property of numeric spaces; vulnerabilities are far more likely to lie in ranges of numbers rather than random numbers as the previously discussed standard approach assumes. Under that approach, test cases of 189, 190 and 191 would provide the same coverage as test cases 80, 180, and 255. In a numeric field, the latter set of test cases should provide more coverage since more sections of the range are tested.

To properly account for numeric spaces in the coverage calculation, a new method was developed that calculates the coverage of a numeric field. Some assumptions were made to make analysis of this kind of space feasible. First, numeric field vulnerabilities are assumed to have specific vulnerability spaces that are contiguous in the invalid space. That is if the values 4 and 6 trigger a vulnerability, then a value of 5 will also trigger that vulnerability. If a vulnerability does not meet this criteria, if it is made up of a few disjoint sequences of values for example, it can be treated as multiple contiguous vulnerabilities. Second, the invalid space itself is assumed to be numerically contiguous. For example, if the value of 3 is invalid and the value of 8 is invalid, then the values 4, 5, 6, and 7 must all also be invalid. A space that does not meet this criterion is said to be *disjoint*. Disjoint numeric field invalid spaces can be handled by computing coverage for each disjoint section as if each were the only invalid space in the field, and then averaging the coverage values to reach a field coverage value for the field as a whole.

The estimated coverage curve equation derivation is very different for numeric

fields than for the non-numeric fields assumed in the derivation of (24) and (29). In the non-numeric case, the value of an arbitrary test case did not affect the coverage calculation, all values had the same probability of discovering a vulnerability. In numeric spaces, a value near a value already tested has a lower probability of discovering a vulnerability than a test case removed from the previously tested values. For this reason, the numeric coverage equation's derivation has to start with a new mathematical representation of a vulnerability.

Since numeric vulnerability spaces are contiguous in invalid space, they can be defined by two values:

v: the lowest value that triggers the vulnerability

l: the length of the vulnerability (note that $l = |V|$).

The probability that a vulnerability exists with specific parameters v and l can therefore be defined by a two dimensional random variable, X. This random variable maps numeric field specific vulnerability spaces to their defining values v and l. The PMF of X, denoted $f_X(v, l)$, is a function that gives the probability a vulnerability has characteristics v and l and can be best defined using research into real vulnerabilities. A good approximation of this function is derived in Section 3.3.

A useful function G can be defined which returns a 1 if the value of a test case, t_i falls in the specific vulnerability space of a vulnerability defined by v and l:

$$G(v, l, t_i) = \begin{cases} 1 & v \leq t_i \leq v + l \\ 0 & \text{otherwise} \end{cases} \tag{31}$$

With this function and the PMF defined, the probability of the first test case discovering any arbitrary vulnerability, V_j, can be found by summing across the PMF for the range that includes the test case.

$$Pr(T_1 \subseteq \tilde{V_j}) = \sum_{v,l} G\left(v, l, t_i\right) f_X\left(v, l\right) \tag{32}$$

Note the subtle difference between t_i and T_i. t_i is the value of test case i, while T_i is a set with only one value, t_i. It would be correct to say $T_i = \{t_i\}$ or even that $T_1 \subseteq \tilde{V_j}$ is equivalent to $t_1 \in \tilde{V_j}$.

When a test case is not first, it will only discover the vulnerability if that vulnerability has not been discovered by a previous test case. To account for this a new function, H, is defined that is 1 when test case i is not in the space of a vulnerability defined by v and l, and 0 when it is in the vulnerabilities range.

$$H\left(v, l, t_i\right) = 1 - G\left(v, l, t_i\right) \tag{33}$$

If no prior test case has tested the space described by v and l, then the product of this H function for all previous test cases will be 1. If any one of the prior test cases did test this space, the product will be zero. This allows it to be incorporated with the G function and f_X to find the probability test case i, where $i > 1$, will discover vulnerability V_j:

$$Pr(T_i \subseteq \tilde{V_j}) = \sum_{v,l} \left(G\left(v, l, t_i\right) \left(\prod_{k=1}^{i-1} H(v, l, t_k) \right) f_X\left(v, l\right) \right) \tag{34}$$

Using (16) for coverage based on the probability each test case will discover a vulnerability, coverage for a field with m test cases $t_1, t_2, \ldots, t_m$ is:

$$C_m = \frac{1}{N} \sum_{i=1}^{m} \sum_{n=1}^{N} Pr(T_i \subseteq \tilde{V_n}) \tag{35}$$

Notice that the right side of (34) does not depend on j. This is because no matter the specific vulnerability's number, it is still model by the same random variable X.

Therefore $Pr(T_i \subseteq \tilde{V}_n)$ can become some function $Q(i)$ that depends on i but not j. That is:

$$Q(i) = \sum_{v,l} \left(G\left(v, l, t_i\right) \left(\prod_{k=1}^{i-1} H(v, l, t_k) \right) f_X\left(v, l\right) \right) \tag{36}$$

Using $Q(i)$ in (35) allows for a simplification:

$$C_m = \frac{1}{N} \sum_{i=1}^{m} \sum_{n=1}^{N} Q(i) = \sum_{i=1}^{m} Q(i) \left(\frac{1}{N} \sum_{n=1}^{N} 1 \right) = \sum_{i=1}^{m} Q(i) \tag{37}$$

Then the right side of (36) can be substituted into (37) to arrive at a complete expression for the coverage, C_m. The limits on the sums are added here for completeness. The only inputs needed to compute the coverage curve are the invalid space I, and the input test cases $t_i,\ i = 1, \ldots, M$ where M is the number of test cases.

$$C_m = \sum_{i=1}^{m} \sum_{v=min(I)}^{max(I)} \sum_{l=1}^{|I|-v+min(I)} \left(G\left(v, l, t_i\right) \left(\prod_{k=1}^{i-1} H(v, l, t_k) \right) f_X\left(v, l\right) \right) \tag{38}$$

Equation (38) is computationally expensive as written due to the number of sums and products, but it can be replaced by a simple algorithm that arrives at the same result. To make this possible a two-dimensional boolean array is created. The elements in this array represent if the corresponding v, l pair has been tested. The domain of this array is the same as the domain of f_X, that is all valid values of v and l. This array will be denoted $possibility\_tested(v, l)$. The algorithm is as follows:

For every test case i

For all possible v

For all possible l

if $t_i \in [v, v + l]$ and $possibility\_tested(v, l) = false$

then $possibility\_tested(v, l) = true$

end for

end for

if last test case, $i = m$

then return $\sum_{v,l} possibility\_tested(v,l) f_X(v,l)$

end for

3.1.3.3 Values Likely to Discover a Vulnerability.

In some cases it may be simple to identify specific values or ranges of values in the invalid space of a field that appear more likely than the others to trigger a vulnerability. Consider if a vulnerability has been discovered in some previous test of the same protocol on a different system. The values that trigger this vulnerability in that test, will likely also trigger a vulnerability in this new test. The tester can input information about which values in a field are more likely to discover a vulnerability, and an estimate of how much more likely. This way, the coverage calculation can accurately reflect the information known by the tester.

To illustrate how this type of information can be incorporated into the coverage calculation, consider a field with an invalid space of $I = \{6, 7, 8, 9, 10\}$ so that $|I| = 5$. The tester has a test set that tests each of these values sequentially so that $t_1 = 6$, $t_2 = 7$, etc. The tester also suspects that a value of 8 is K times more likely to discover a vulnerability than otherwise thought. The tester wants to know what the coverage of the test set is after the applying each test case.

Without the suspicion of value 8, the coverage each test case provides can be found using (29). This equation provides the cumulative coverage for each test case, but the specific coverage each test case adds to the total can also be found by dropping the leading summation. This value can be denoted c_i where i is the test case index.

$$c_i = \sum_{x=1}^{|I|} f_X(x) R(x,i) \tag{39}$$

Using the PMF from Section 3.3, specifically equation (80), the individual coverage contribution of each of the five test cases can be computed. This result is shown in Figure 9. Like any complete set of coverage values, these five coverage values sum to one because when every value in the invalid space has been tested, the space is 100% covered.

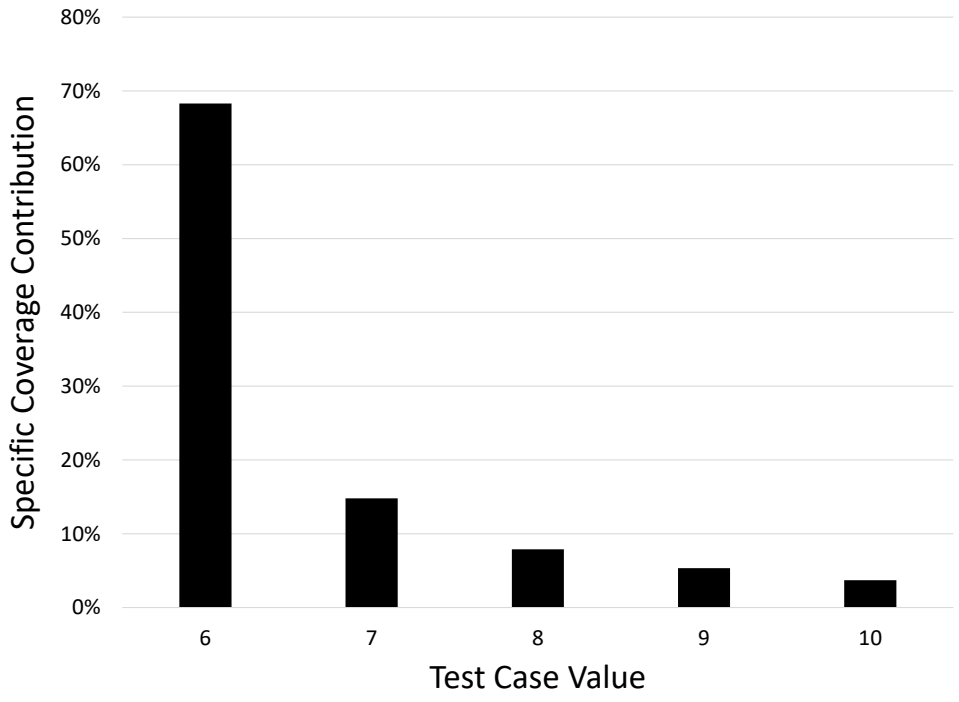

Figure 9. The specific coverage, c_i, provided by each sequential test case for the example field without value weighting

To incorporate the additional information about the value of 8 in test case 3, the coverage of this test case, c_i, needs to be increased relative to the other test cases. The sum of all coverage values must remain at 1 however, so in order to add coverage to test case 3, coverage provided by the rest of the test cases must be reduced. Since the relationships between these test cases ought to be maintained, all of them must

be reduced by some factor r, where $0 < r < 1$. These new reduced coverage values will be called c'_i with the definition $c'_i = r \cdot c_i$.

The coverage for test case 3, c_3, undergoes the same reduction, but then the additional information that this test case should provide K times the coverage than it otherwise would can be applied so that $c'_3 = K \cdot r \cdot c_3$. As stated before, the individual coverage values must sum to 1; this property allows r to be computed in terms of the original coverage values as shown:

$$\sum_{i=1}^{5} c'_i = 1 \tag{40}$$

$$\left(\sum_{i=1}^{5} r \cdot c_i\right) + (K-1) \cdot r \cdot c_3 = 1 \tag{41}$$

$$r\left(\left(\sum_{i=1}^{5} c_i\right) + (K-1) \cdot c_3\right) = 1 \tag{42}$$

$$r\left(1 + (K-1) \cdot c_3\right) = 1 \tag{43}$$

$$r = \frac{1}{1 + (K-1) \cdot c_3} \tag{44}$$

This approach can be applied in general so that many values can provide more coverage based on tester knowledge. To do this, every test case t_i can have some weight K_i. If the tester has no information about a value, a test case that takes on this value receives a weight of 1. The reduction factor r can be calculated by assuming some test set of $|I|$ test cases that contains each value in the invalid space once. Then, by the same method used in the previous example, an expression for r can be found that depends on the value weights, K_i, and the coverage each value provides without weighting, c_i.

$$\sum_{i=1}^{|I|} c'_i = 1 \tag{45}$$

$$\sum_{i=1}^{|I|} K_i \cdot r \cdot c_i = 1 \tag{46}$$

$$r = \frac{1}{\sum_{i=1}^{|I|} K_i \cdot c_i} \tag{47}$$

The new coverage values, c'_i can be found simply as:

$$c'_i = r \cdot K_i \cdot c_i \tag{48}$$

Noting again the expression for c_i shown in (39), a complete expression for c'_i can be found:

$$c'_i = r \cdot K_i \sum_{x=1}^{|I|} f_X(x) R(x, i) \tag{49}$$

And applying this to the original field coverage equation, (29), a complete expression for standard coverage with weighting can be found:

$$C_m = \sum_{i=1}^{m} r \cdot K_i \sum_{x=1}^{|I|} f_X(x) R(x, i) \tag{50}$$

Returning to the example, the additional information about the test case of 8 can be incorporated to reach a better specific coverage value contributed by each test case. Using the equations just derived with a $K_3 = 5$, r for this field is found to be 0.76. This leads to new, scaled specific coverage contributions of each case shown in Figure 10 contrasted with the original specific coverage values shown in Figure 9.

For numeric fields, the same weighting approach can apply by noting that a different expression for c_i can be extracted from (38):

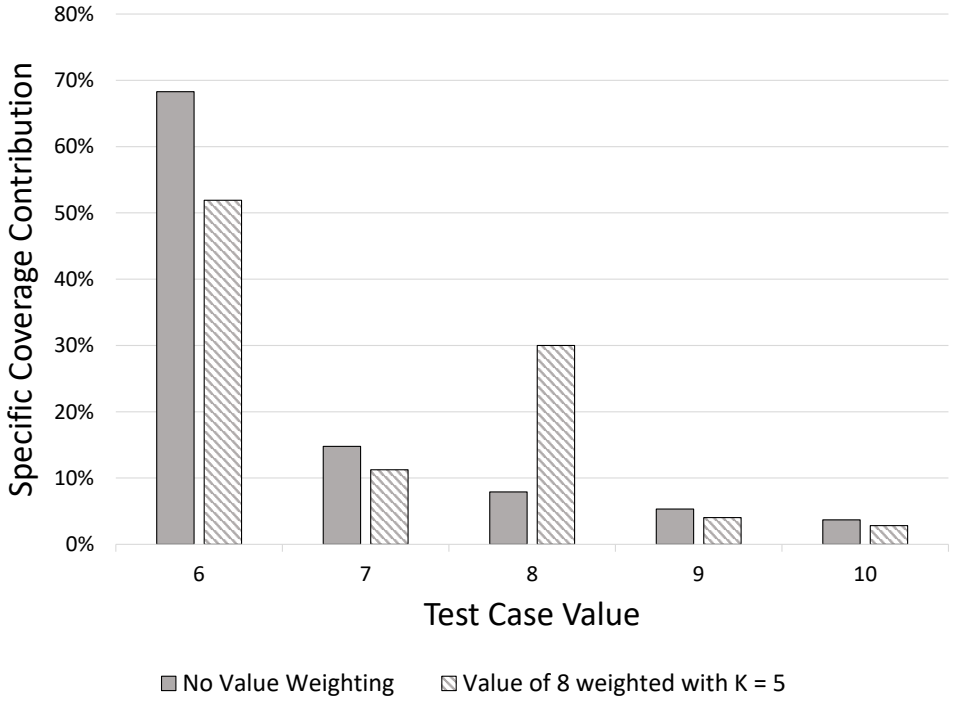

Figure 10. The specific coverage, c_i, provided by each sequential test case for the example field with and without value weighting

$$c_i = \sum_{v=min(I)}^{max(I)} \sum_{l=1}^{|I|-v+min(I)} \left(G\left(v,l,t_i\right) \left(\prod_{k=1}^{i-1} H(v,l,t_k) \right) f_X\left(v,l\right) \right) \tag{51}$$

This leads to the weighted numeric field coverage equation:

$$C_m = \sum_{i=1}^{m} r \cdot K_i \sum_{v=min(I)}^{max(I)} \sum_{l=1}^{|I|-v+min(I)} \left(G\left(v,l,t_i\right) \left(\prod_{k=1}^{i-1} H(v,l,t_k) \right) f_X\left(v,l\right) \right) \tag{52}$$

Shortcomings of this Method. This method was implemented in ExFuzz for any number of weighted inputs per field. During the implementation it was discovered that the method presented was somewhat incomplete, requiring ExFuzz to

add some additional procedures to the method. It was also found that a contradiction arises with other assumptions about the coverage metric when this method is applied.

First, the calculation of r in (44) requires knowledge of all K_i and c_i with $i = \{1, 2, \ldots, |I|\}$. Rarely in a coverage calculation is every one of these values computed. In fact, the only time they are is when the test set contains a test case for every value in the invalid space (in this case $m = |I|$). When this happens, field coverage is always 100% regardless of any value weighting.

To calculate the missing c_i and their associated K_i, ExFuzz completes the computation of specific coverage values for all i. To do this it has to select the order in which the remaining, untested, invalid values are incorporated into the calculation. It selects the values with the highest weights first, with ties going to the lower decimal value. This approach was selected because, of any approach, it assigned the smallest coverage to any test set that did not include weighted values.

This addition to the method highlighted the second flaw. Since the sequence of the test cases could be manipulated to provide more coverage, coverage was dependent on the arrangement of the test cases in a test set. This meant that two identical test sets could provided different amounts of coverage if test cases were arranged in a different order. Nothing in this thesis's definition of coverage implies that this should be possible. The number of vulnerabilities a test set discovers should be independent of the arrangement of the test cases in the stateless SUT model assumed.

Possible Solutions for Future Work. While there was not enough time to implement a solution to these problems, use of the value weighting feature of ExFuzz was avoided for all of the results in Chapter V. The tool would certainly be stronger with this feature, so some time was spent thinking about how to address these issues. Two possible solutions have come to mind.

- Redefine K_i and c_i so that their index, i, is not a representation of the test case number, but of its placement based on weight. Consider the example field used in this section, under the current method $K_3 = 5$ while all other K's are 1. Instead, order the test cases and their accompanying K weights by weight so that $K_1 = 5$, t_3 becomes t_1 and the other test cases are assigned t_2 to t_5 in order of their decimal value. This removes the dependence on test case order and increases coverage for test cases that include weighted values over the current method.

  This approach doubles down on the solution implemented by ExFuzz to address the incompleteness of the current method. It will be computationally expensive, but addresses the contradiction with this thesis's definition of coverage.

- Address weighted values earlier in the coverage calculation. Under the current method, weighted values result in scaling the specific coverage contributions of each test case, leaving the internals of the coverage calculation alone. Weighted values could be incorporated earlier however. There is no obvious way to do this, but one potential approach could involve expanding the notional invalid space used in the calculation.

  For example, the invalid value of 8 in the example can be treated as five different invalid values. This would increase $|I|$ from five to nine. When the value of 8 is tested, it would count as testing five values simultaneously. This approach would bypass the modifications that led to the contradiction with the coverage definition, and also move away from the current flawed method. However, this method would distort the application of the PMF which is not based on artificially enlarged invalid spaces.

3.2 Test Set Coverage

This section discusses how to apply the field coverage metric derived in Section 3.1 to many test cases for many different fields to compute the coverage of a test set. First the method to arrive at test set coverage for first order vulnerabilities will be discussed in Section 3.2.1. Then method to define coverage for multi-order vulnerabilities will be presented in Section 3.2.2, accompanied by a discussion on handling multi-order structure field combinations in Section 3.2.3. This section concludes by showing how to combine the coverage values of different orders into a single coverage value for the fuzz test in Section 3.2.4.

3.2.1 First Order Coverage.

The coverage criterion presented in this chapter assumes that a protocol can be represented as a finite set of fields. A method for modeling a protocol in such a fashion has been developed and is based on the Peach Fuzzing Platform [10]. This model, called a data model, is discussed in Chapter IV. According to the data model, not every field must be present in a test case, but every bit in a test case must belong to some field. A test case, one single message sent to the SUT, can be represented by a data model as a series of fields with values. As shown in Chapter IV, a single message may contain a field that is repeated multiple times, or it may not contain certain fields at all. The rules of which fields can appear when and how many times they may appear are all contained in the data model.

As an example, consider a protocol with three fields, A, B and C, each three bits long. Field A is always first, and dictates how many fields will follow in the message. Field B is optional and may be repeated up to 6 times. The value of 111, or decimal 7, is not allowed in field B. Field C is always the last field and is required to be 111, or 7. Since at least field C is required, field A may not be 000.

Consider the test case 000111010011. Field A contains the invalid value of 000, while the actual number of fields following it is 3. Field B occurs twice, the first time it is the invalid value of 111, but the second time it is the valid value 010. Field C occurs once at the end and is invalid since it is not the required value of 111. It is tempting to assume that this test case provides coverage for all three fields because it tests values in each of the field's invalid spaces. This assumption would be a mistake however. If the SUT were to immediately drop any message with 000 in the first field, the effects of placing 011 in field C would not be known. Likewise, if the SUT immediately drops messages without the terminating 111 string, the effect of placing 000 in field A would not be known. Without knowledge of the SUT's behavior, this test case provides no first order coverage. It would however provide third order coverage as will be discussed later.

This example gets at a fundamental idea about fuzzing. Inputs must be valid enough to be processed by the SUT, but invalid enough to trigger a vulnerability. These inputs are termed semi-valid inputs and are discussed in depth in [26]. Because of this observation, in order for a test case to provide coverage for a first order vulnerability, all included fields but one must have a value in their fields valid space. In addition to this requirement, a test case will not provide any first order coverage if it contains a series of fields whose values, when taken together, are considered invalid. These combinationally invalid fields are called Invalid in Combination field sets, or IIC sets, and are discussed with examples later in this chapter. For now, it is enough to know that if one exists in a test case, the test case cannot provide first order coverage.

A test case could also consist of only valid values. By axiom four of fuzz testing, presented in Section 2.2.3, this test case cannot increase the coverage of the test set. Therefore, only the test cases where one and only one field is invalid provides first

order coverage.

Another important consideration is that any field with no invalid space cannot be assigned a field coverage value per the method presented in Section 3.1. Without an invalid space, a vulnerability space cannot exist, so these fields are not included in the test set coverage calculation.

These limitations on which test cases and fields may provide first order coverage can be summarized by the following two limitations:

- **First Order Test Case Limitation:** A test case may provide first order coverage if and only if it contains exactly one field with an invalid value and contains no Invalid in Combination (IIC) sets.
- **First Order Field Limitation:** A field is assigned a coverage value and incorporated into the final first order coverage metric, $\mathbb{C}_1$, if and only if the cardinality of its invalid space is non-zero, $|I| \neq 0$.

Considering only the test cases and fields that meet these limitations, the coverage a test set provides can be computed. This can be done by following these steps:

1. Identify the field with an invalid value for every test case in the test set
2. Sort the test cases into groups based on which field is invalid
3. For each group, calculate field coverage using the appropriate field coverage equation; see Table 2
4. Combine field coverage values to find first-order coverage

The first two steps are self explanatory. For step 3, the appropriate field coverage equation must be selected from Section 3.1 based on any special information known about the field. Table 2 shows which field coverage equation to used based on the characteristics of the field.

Table 2. Field coverage equation lookup table

Is the field numeric?	Does the field contain weighted values?	Then use eq.	on page
no	no	(29)	47
yes	no	(38)	55
no	yes	(50)	59
yes	yes	(52)	60

To implement step four, an approach had to be found that combined field coverage values in some way to produce a coverage value for the entire test set. To determine the appropriate way to do this, the definition of test set coverage was analyzed. Per the definition, the final metric needed to be a measure of the expected percentage of first order vulnerabilities found by test cases in the test set. This metric, $\mathbb{C}_1$, was derived as follows:

Let F be the number of fields in a protocol where the size of the invalid space, $|I| \neq 0$.

Let $C_i, i = 1, 2, \ldots, F$ be the coverage of i^{th} field where $|I| \neq 0$.

Let N_i be the number of vulnerabilities present in field i, and:

$$N_{Total} = \sum_{i=1}^{F} N_i \tag{53}$$

Let L_i be the number of vulnerabilities in field i discovered by the test set. Then by the definition of field coverage shown in (3):

$$C_i = \frac{E\left[L_i\right]}{N_i} \tag{54}$$

The expected total number of vulnerabilities found, V_{Found}, can then be shown to be:

$$E\left[V_{Found}\right] = \sum_{i=1}^{F} E\left[L_i\right] = \sum_{i=1}^{F} C_i N_i \tag{55}$$

Therefore, test set first order coverage $\mathbb{C}_1$, the expected percentage of all first order vulnerabilities can be found as:

$$\mathbb{C}_1 = \frac{E\left[V_{Found}\right]}{N_{Total}} = \frac{1}{N_{Total}} \sum_{i=1}^{F} C_i N_i \tag{56}$$

This definition of test set coverage depends on knowing exactly how many vulnerabilities exist and what fields they are in. If this was known, the fuzz test would be unnecessary, so the dependence on N_i must be removed. Equation (56) shows that the proper calculation of first order test set coverage is an average of field coverage values weighted by the number of vulnerabilities present in each field. To remove this dependence, N_i can be replaced by the expected number of vulnerabilities present, $E\left[N_i\right]$:

$$\mathbb{C}_1 = \frac{1}{\sum_{i=1}^{F} E\left[N_i\right]} \sum_{i=1}^{F} C_i E\left[N_i\right] = \frac{1}{E\left[N_{Total}\right]} \sum_{i=1}^{F} C_i E\left[N_i\right] \tag{57}$$

This value depends instead on what is known about the field: at least the size of its invalid space, and potentially any more information the tester manually enters into a fuzzer about the field. $E\left[N_i\right]$ can be estimated by considering all of the information available about a field and by analyzing the distribution of vulnerabilities in tested protocols.

In the simplest case, the fuzzer has no information about the nature of any field that would cause $E\left[N_i\right]$ to be different for any two values of i. In this case $E\left[N_i\right] = E\left[N_1\right]$ for all i so:

$$\mathbb{C}_1 = \frac{1}{\sum_{i=1}^{F} E\left[N_1\right]} \sum_{i=1}^{F} C_i E\left[N_1\right] = \frac{E\left[N_1\right]}{F \cdot E\left[N_1\right]} \sum_{i=1}^{F} C_i = \frac{1}{F} \sum_{i=1}^{F} C_i \tag{58}$$

Which is simply the average of the field coverage values. If more information is provided to the fuzzer about the nature of a specific field that increases the likelihood

of a vulnerability being present, a weighted average can be used. For example, if a tester believes field six is four times as likely to have a vulnerability because she has knowledge of how other similar systems have handled this field, she can assign $E[N_6] = 4$. Then by default, $E[N_i] = 1$ for all $i \neq 6$. In this case the weighted definition for first order test set coverage in (57) is used instead of the simple unweighted one in (58).

3.2.2 Nth Order Coverage.

3.2.2.1 Types of Multi-Order Vulnerabilities.

Vulnerabilities are not always confined to a single field. For example, consider two fields, A and B, each 5 bits. Field A is an indicator of the priority of the information contained in field B. The values 0 to 20 are valid, with 0 being the lowest priority and 20 being the highest. Values above 20 are invalid. Field B is the field from Figure 4, with the same vulnerable inputs. In this example however, the application receiving the message with these two fields is only interested in the value of B if its priority is 19 or 20. Therefore, the vulnerabilities laid out in Section 3.1.1 for field B are not discovered unless the value of A is 19 or 20. Since the presence of this vulnerability is dependent on two fields, it is said to be a second order vulnerability.

A second order vulnerability, or, more generally, a multi-order vulnerability, exists in the invalid space of a field combination, instead of the invalid space of a field. These spaces can be depicted in a similar manner to the first order spaces, but in these spaces each element represents a unique combination of values, rather than a single unique value. The spaces associated with this example are shown in Figure 11.

In this depiction of a second order space, each value, or square, actually represents a set of two values: the value of field A and the value of field B. Any value set where at least one value is invalid is in the invalid space of this field combination. The

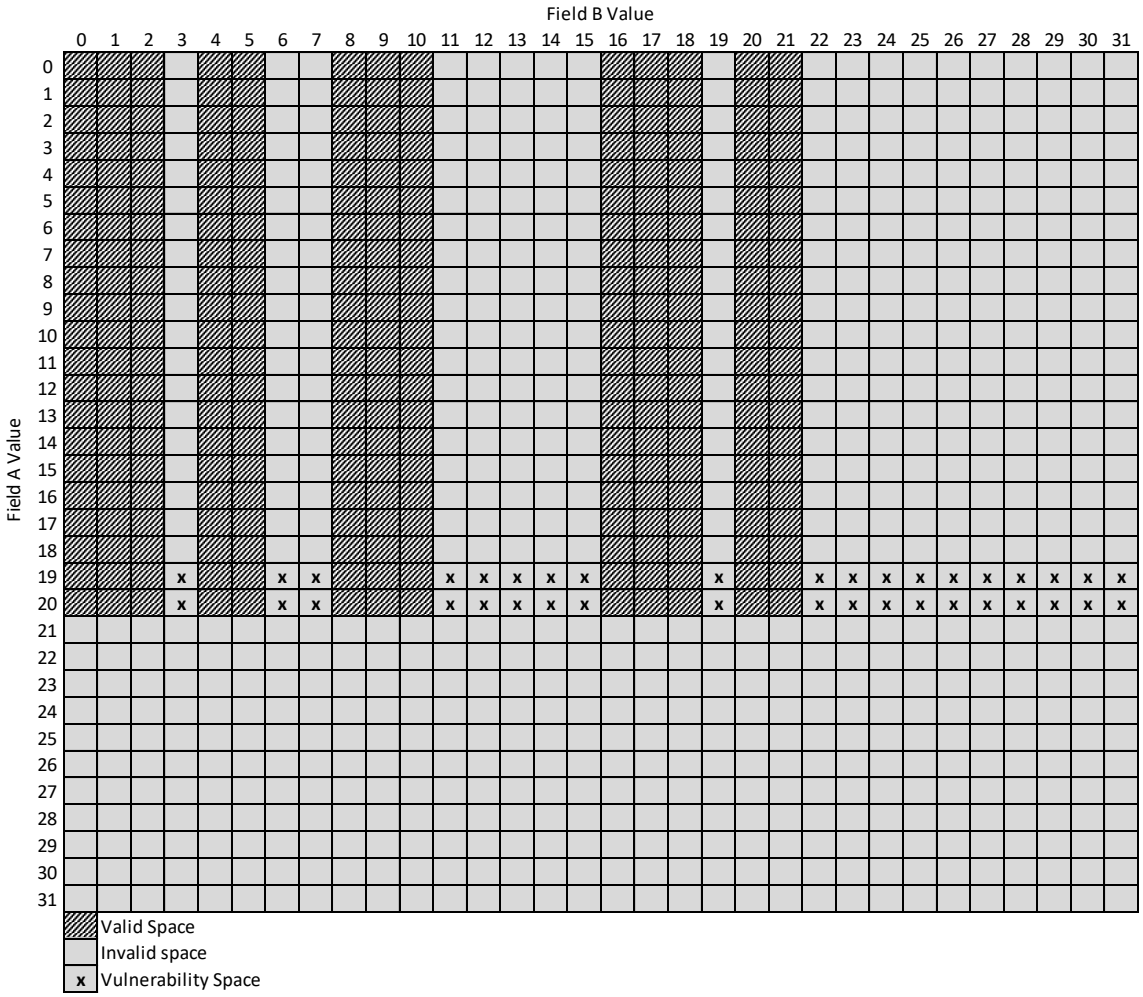

Figure 11. A representation of the second order spaces associated with two five bit fields

vulnerability described is comprised of value pairs where field A is always valid (either 19 or 20) and field B is always invalid.

There is also the possibility a vulnerability lies in the space where field A is greater than 20 and field B is invalid so that both fields are invalid. If a vulnerability is triggered only if both fields have a value of 31 then it would lie in this part of the invalid space.

Another important case to consider is two values for two different fields that are valid on their own, but become invalid when combined. As a simple example, consider two fields where the first, a four bit field, represents the month of the year and the

second, a five bit field, represents the day of the month. Obviously month values of 0, 13, 14 and 15 are invalid, and the day value of 0 is invalid in the single field case. When combined however, February 30th and 31st are invalid, even though each value is valid for its field. This combination is shown in Figure 12. The tester will generally have to enter these types of invalid combinations manually in the protocol data model. These pairs of fields and values will be called invalid in combination (IIC) pairs and make up another type of multi-order vulnerability.

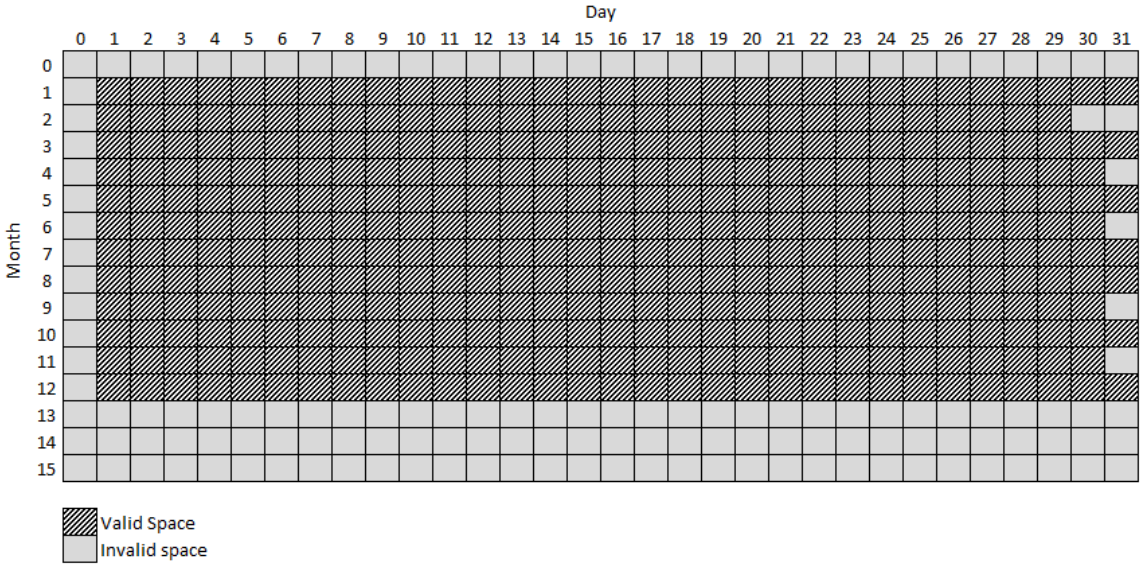

Figure 12. A representation of the second order spaces associated with day and month fields

The possible types of second order vulnerabilities are summarized as:

- **One field invalid, one field valid.** Ex: When field A is 19 or 20 and field B is any invalid value a vulnerability is triggered.
- **Two invalid fields.** Ex: When field A is 31 and field B is any invalid value a vulnerability is triggered.
- **Two valid fields that are IIC.** Ex: A month field denoting February and a day field with a value of 30 or 31 trigger a vulnerability.

It is important to note that a vulnerability may combine types. Consider if the same vulnerability shown in figure 11 was also triggered when field A had a value of 21. Then some triggering inputs would have one valid and one invalid input while others would have 2 invalid inputs.

Multi-order vulnerabilities may exist as any of the types presented here or as combinations and extensions of them. Perhaps a vulnerability is triggered only when field A is 19 or 20, field B is invalid and the reported month and date are IIC. Since this vulnerability depends on the values of four different field, it is a fourth order vulnerability.

3.2.2.2 Computing Multi-Order Coverage.

To compute second order test set coverage $\mathbb{C}_2$, the same formulas used for $\mathbb{C}_1$ can be applied with some modification. For first order coverage, coverage values for each field were found and then combined to reach a test set coverage value. Instead of using fields, second and higher order coverage is found by looking at field combinations. For example, consider a protocol with three fields, A, B, and C, each with a non-empty invalid space. The coverage of fields A, B and C would be combined using (57) or (58) to find $\mathbb{C}_1$. The coverage of field combinations AB, AC, and BC would be combined in a similar manner to find $\mathbb{C}_2$. Third order coverage, $\mathbb{C}_3$, would be the same as the coverage of field combination ABC.

The process to determine the R^{th} order coverage of a test set, $\mathbb{C}_R$, is very similar to the process to find $\mathbb{C}_1$. As in that process, there are limitations on the test cases and fields, in this case field combinations, which are considered. For the case of order R:

- **R^{th} Order Test Case Limitation:** A test case may provide R^{th} order coverage if and only if there exist some combination of R fields which has an invalid value

set and contains all the invalid fields and Invalid in Combination (IIC) field sets present in the test case.

- **R^{th} Order Field Combination Limitation:** A field combination is assigned a coverage value and incorporated into the final R^{th} order coverage metric, $\mathbb{C}_R$, if and only if the cardinality of its invalid space is non-zero, $|I| \neq 0$.

Considering only the test cases and field combinations that meet these limitations, the R^{th} order coverage a test set provides can be computed. This is done by following a similar set of steps to that used for first order coverage.

1. For each test case in the test set identify the field or fields with invalid values and all IIC values sets
2. For each combination of R fields, collect all test cases where every invalid field and IIC value set are entirely contained in fields in that combination
3. For each group, calculate field combination coverage
4. Combine field combination coverage values to find R^{th} order coverage

The first two steps simply involve arranging the test cases to identify which test cases provide coverage to which field combinations. It should be noted that, unlike for first order field coverage, test cases can contribute to the coverage of multiple field combinations. If within a test case field A is the only invalid field and it appears with a valid field B and a valid field C, then both field combinations AB and AC are invalid in this test case and receive some coverage. Table 3 summarizes the first and second order coverage a test case can provide based on the number of invalid fields and IIC value pairs it has.

Table 3. Possible coverage contributions for test cases with different arrangements of invalid fields and IIC pairs.

If a test case has:	Then it provides:	
	First order coverage	Second order coverage
One invalid field	For one field	For many pairs of fields
Two invalid fields		For one pair of fields
No invalid fields and no IIC pairs		
No invalid fields and one IIC pair		For one pair of fields
No invalid fields and 2+ IIC pairs		

The third step is achieved by treating the field combination and its associated spaces like a first order field and finding coverage using (29). The numeric coverage equations are not directly applicable since multi-order spaces are not linear as is required for numeric spaces, but in the future, field combination coverage for combinations of numeric fields may be developed. The PMF applied in the field coverage equation can be modified to reflect the fact that a multi-order field's coverage is being computed. The proper PMF to use would have to be determined by studying multi-order vulnerabilities. For this thesis the PMF that was already derived for first order spaces was simply applied to multi-order spaces as well.

The fourth step is also similar to the first order steps on page 65, but the derivation of the equation used to combine field combination coverage values is a bit trickier. This is because the number of fields with an invalid space, F, now needs to become the number of field combinations with an invalid space, a slightly more challenging figure to find.

The number of field combinations with non-empty invalid spaces, call this M, can be computed using F, the total number of fields in the protocol, F_I, the total number of fields with invalid spaces in the protocol, and FC_{IIC_R} the number of order R field combinations that contain IIC values, but have empty first order invalid spaces for all

fields in the combination. Notice that the number of field combinations with at least one field with an invalid space is the same as the total number of field combinations minus the valid only combinations. Adding in the IIC containing field combinations, M can be computed as:

$$M = \binom{F}{R} - \binom{F - F_I}{R} + FC_{IIC_R} \tag{59}$$

Upon implementation of this equation, it was found that it counted one case that should not have been. In some more complex protocols, certain fields may never validly appear in the same message. The field combinations of these sets of fields can never exist with all other fields being valid, so their field combination should not be included in the number of total combinations. Let the total number of such field combinations for an order R be Q_R. Then (59) becomes:

$$M = \binom{F}{R} - \binom{F - F_I}{R} + FC_{IIC_R} - Q_R \tag{60}$$

To help notate the equations related to multi-order coverage, a function G is defined. G maps every possible combination of R fields — $i_1, i_2, \ldots, i_R$ — to a unique integer p. So $G(i_1, i_2, \ldots, i_R) = p$. An inverse function H is also defined that maps the integer p back to a set of R fields, $H(p) = \{i_1, i_2, \ldots, i_R\}$. Using this H function and assuming no weighting of any field combination, R^{th} order coverage can be found as the average of all R^{th} order field combination coverage values:

$$\mathbb{C}_R = \frac{1}{M} \sum_{p=1}^{M} C_{H(p)} \tag{61}$$

Note here that $C_{H(p)}$ or equivalently $C_{\{i_1,i_2,\ldots,i_R\}}$ is a field combination coverage computed in step three above. Similarly for weighted field combinations with $N_{H(p)}$ representing the number of vulnerabilities present in the field combination defined by

$H(p) = \{i_1, i_2, \ldots, i_R\}$ for some fields $i_1, i_2, \ldots, i_R$:

$$\mathbb{C}_R = \frac{1}{E\left[N_{Total}\right]} \sum_{p=1}^{M} C_{H(p)} E\left[N_{H(p)}\right], \qquad \text{where } N_{Total} = \sum_{p=1}^{M} N_{H(p)} \tag{62}$$

3.2.3 Structure Fields.

As they did in Section 3.1, structure fields require special attention with regard to multi-order spaces. The two, yet unmentioned, types of structure fields used in this thesis are very similar and relate to the choice element presented in Chapter IV. When combined with the repeating relation field described in Section 3.1, they require special definitions of their valid and invalid field combination spaces.

3.2.3.1 Choice Relations.

The first type of structure field is called a choice relation. This type of field is a custom peach pit extension and uses the relation element that is also used by the repeating relation structure field. The value in a choice relation dictates the choice that some choice element in the data model will make. For example, consider three fields, A, B and C. Field B represents a distance in feet; field C represents a distance in meters. Field A is a single bit; if it is a 0, distance should be represented in feet and if it is a 1, distance should be represented in meters. The data model describing this example would have fields B and C within a choice and field A outside of this choice with a relation to the choice.

It is tempting to assume that "distance" should be represented by only one field whether it is interpreted as feet or meters, negating the need for a choice at all. However, a field that represents distance in feet may have a different invalid space than one that represents it in meters, so only by treating them as separate fields can coverage be accurately computed. In this example, the coverage of the choice

relation field, field A, can computed in the same way as non-structure fields. It requires mentioning here however because it behaves differently when it is part of a field combination with a repeating relation field.

3.2.3.2 Token Fields.

The second type of structure field is called a token field. This field resides inside a choice, and has a fixed value that indicates the selected choice. Each possible choice contains its own token field with a unique fixed value. For example, consider an eight bit block of code that can be defined in two different ways; either field A - 2 bits, field B - 4 bits and field C - 2 bits, or field D - 2 bits, field E - 6 bits. In order for the SUT to determine which interpretation of these 8 bits to apply, it looks at the first two bits. If they are 01, then the fields A, B, C interpretation is used, if they are 10, then the fields D, E interpretation is used.

In this example fields A and D are token fields. Notice that only 01 is valid for field A and only 10 is valid for field D. Further, field A can never take a value of 10, because if it did it would be interpreted as field D by the SUT. Likewise, field D cannot take on the value 01, or it would be interpreted as field A. If those two bits had a value of 00 or 11, it would be impossible to claim that the field was either field A or D. However, all bits must belong to a field for this coverage criterion, so for purposes of the coverage calculation, one token is assigned the title of the leading token. Any value that does not match a token is considered an invalid instance of the leading token, and the fields that follow (B and C in this example) are ignored for purposes of calculating coverage. This leading token definition causes each non-leading token to have an empty invalid space, excluding it from the coverage calculation. Thus, all first order coverage of a token field is done through the leading token, and can be found in the same way as any non-structure field coverage is found since the invalid

space is structured in the same way.

Data models that use token fields can often represent the same protocol by using choice relations instead. The only time when it may be easier to use token fields is when the token field is not the first field in the choice, but this is rarely the case. Token fields are included in this thesis because, unlike choice relations, they are a part of the peach pit language, and were the only type of structure field that can be associated with a choice before support was added for choice relations in the modified peach pit [10].

3.2.3.3 Combining Structure Fields.

A unique situation arises in a data model when a choice element is inside a repeating block. The result is the choice gets made multiple times and there are multiple instances of token or choice relation fields. In these cases, a protocol often defines rules about which choices can be made when in the sequence of repeats. These rules can be captured by the Rule Set extension to the peach pit language which is discussed in Chapter IV. The Restaurant Locator Protocol (RLP), which is described in Section B.3, provides an example of this.

The RLP states that Word 1 must be the first word, and Words 2 and 3 must exist in a pair. The meeting or breaking of these rules is determined by a combination of the value of the token field choosing the word and its place in the repeating sequence. These limitations can therefore be thought of as affecting the field combination of the repeating relation field and the token (or choice relation) field. A sequence of choices that violates these rules would not contain any invalid values, and it should therefore be treated as an IIC field set.

To visualize the valid and invalid spaces of a field combination between a repeating relation field and a choice relation or token field, a simple case can be looked at.

Consider a repeating relation field only 2 bits long which relates to a block that repeats 1 to 4 times. The adjustment of this relation is +1 so that a value of 0 in the field means the block repeats 1 time. There are only two possible choices, A and B, represented by a 1 bit token in each choice. A rule set is defined that says all choice A's must come before B's. The valid and invalid spaces for this field combination are shown in Figure 13. The size of the invalid space in this case would be 106 combinations of values.

Using this unique representation of an invalid space when a field combination is of the repeating choice type, and using the previously described method for all other field combinations, multi-order coverage can be computed for a test set. Once all the desired multi-order coverage calculations are performed, it is time to combine these coverage values into a final coverage metric for the entire fuzz test.

3.2.4 Combining Coverage Orders.

Given a test set and a protocol, the coverage of the test set for each order, 1 through the number of fields, F, can be found. A single coverage value for the entire

Actual Repeats	Repeating Relation Value	Repeated Choices																Number of Permutations (with replacment)	Valid	Invalid
1	0	A	B															2	2	0
1	1	A	B															2	0	2
1	2	A	B															2	0	2
1	3	A	B															2	0	2
2	0	AA	AB	BA	BB													4	0	4
2	1	AA	AB	BA	BB													4	3	1
2	2	AA	AB	BA	BB													4	0	4
2	3	AA	AB	BA	BB													4	0	4
3	0	AAA	AAB	ABA	ABB	BAA	BAB	BBA	BBB									8	0	8
3	1	AAA	AAB	ABA	ABB	BAA	BAB	BBA	BBB									8	0	8
3	2	AAA	AAB	ABA	ABB	BAA	BAB	BBA	BBB									8	4	4
3	3	AAA	AAB	ABA	ABB	BAA	BAB	BBA	BBB									8	0	8
4	0	AAAA	AAAB	AABA	AABB	ABAA	ABAB	ABBA	ABBB	BAAA	BAAB	BABA	BABB	BBAA	BBAB	BBBA	BBBB	16	0	16
4	1	AAAA	AAAB	AABA	AABB	ABAA	ABAB	ABBA	ABBB	BAAA	BAAB	BABA	BABB	BBAA	BBAB	BBBA	BBBB	16	0	16
4	2	AAAA	AAAB	AABA	AABB	ABAA	ABAB	ABBA	ABBB	BAAA	BAAB	BABA	BABB	BBAA	BBAB	BBBA	BBBB	16	0	16
4	3	AAAA	AAAB	AABA	AABB	ABAA	ABAB	ABBA	ABBB	BAAA	BAAB	BABA	BABB	BBAA	BBAB	BBBA	BBBB	16	5	11
		Valid, based on Repeating Relation and rule that all A's must come before B's															**Total:**	120	14	106

Figure 13. The valid and invalid space associated with a repeating choice field combination

test set has not been found yet however. To get this, the coverage values for all vulnerability orders must be combined. To do this, either the number of vulnerabilities present of each order ($\mathbb{N}_R$) needs to be known, or this figure needs to be estimated using the expected number of vulnerabilities of each order. This leads to the following definition of coverage for a test set:

$$\mathbb{C}_{Fuzz\ Set} = \frac{1}{E\left[\mathbb{N}_{Total}\right]} \sum_{r=1}^{F} \mathbb{C}_r E\left[\mathbb{N}_r\right], \qquad \text{where } \mathbb{N}_{Total} = \sum_{r=1}^{F} \mathbb{N}_r \tag{63}$$

This definition of coverage is of little practical use however because the higher order space sizes are huge compared to the first and second order sizes. This leads to infeasibly large test sets being needed to provide much coverage of these higher order field combinations. To mitigate this issue, higher order vulnerabilities can be assumed to be very rare or not of interest since they are so hard to find by either a tester or an attacker. Therefore, the weight of higher orders, or the expected number of higher order vulnerabilities can be assigned a value of 0, simplifying the coverage calculation considerably.

The expected number of lower order vulnerabilities present in a protocol implementation can be found by studying real protocols and figuring out the ratio between the number of first, second and third order vulnerabilities. Without any of this research into these ratios, this thesis proposes using one of two fuzz set coverage calculations. The first assumes all vulnerabilities of order four and higher are either non-existent or not important to the fuzz tester. This coverage calculation also assumes that 40% of all vulnerabilities are first order, 40% are second order and 20% are third order. This reflects the fact that many fields in common protocols have a dependence on one other field (latitude and longitude, priority and value, validity and value) and fewer have a dependence on two other fields (hour, minute, second). This simplified coverage equation is therefore:

$$\mathbb{C}^1_{Fuzz\ Set} = \frac{2}{5}\mathbb{C}_1 + \frac{2}{5}\mathbb{C}_2 + \frac{1}{5}\mathbb{C}_3 \tag{64}$$

The second simplified coverage equation is for fuzz testing where testing for third and higher order vulnerabilities is infeasible due to the number of fields, the size of the fields, or the speed at which the test cases can be applied. Since testing for these vulnerabilities is infeasible, the coverage calculation assumes that such vulnerabilities do not exist. This may be a logically poor assumption, but if the tester with the automated fuzzer cant find these higher order vulnerabilities, the attacker likely faces equal difficulty in finding them. This equation assumes that first and second order vulnerabilities are equally likely to exist in the protocol.

$$\mathbb{C}^2_{Fuzz\ Set} = \frac{\mathbb{C}_1 + \mathbb{C}_2}{2} \tag{65}$$

3.3 Approximating Vulnerability Distributions

Section 3.1, estimating field coverage, introduces a probability mass function, $f_X(x)$, which models the nature of vulnerability spaces. This function is used to find field coverage in (29), (38), (50), and (52). For basic, non-numeric, fields, the PMF maps vulnerability sizes to the probability any given vulnerability is exactly that size. For example, $f_X(5) = 0.25$ is equivalent to saying that 25% of specific vulnerability spaces have a cardinality of 5. For numeric fields, a 2-dimensional PMF is used in a similar manner. To reach an accurate coverage value, these functions, what they represent, and how they are calculated needs to be understood. This section describes the development PMFs which were then implemented in the coverage calculator.

The true nature of these PMFs is essentially unknowable. To find it testers would need to discover every vulnerability in every protocol, an accomplishment that would make fuzz testing obsolete. An educated guess can be made instead however. The

closer this guess is to the real nature of the function, the more accurate the coverage calculation will be.

3.3.1 Open Source Protocol Analysis.

The best method of estimating this PMF would be to do a comprehensive study of vulnerability discovery in protocol implementations and to identify the nature of those found vulnerabilities. Unfortunately this would have been a massive undertaking and the time-line for this thesis project would not allow for it. Instead, four open source protocols were looked at, the invalid spaces of many of their fields were identified, and groups of inputs which seemed most likely to cause an issue for a poorly implemented receiver were recorded. The protocols were:

- Global Positioning System — Standard Positioning service (GPS-SPS)
- Bluetooth Network Encapsulation Protocol (BNEP)
- Open Smart Grid Protocol (OSGP)
- Ethernet POWERLINK

For each protocol, fields were selected for inclusion in the PMF estimation effort based on these criteria:

- The protocol specification clearly indicates some values are not valid. This allows the field to have an invalid space and potentially vulnerabilities.
- The meaning of the values of the field are clear and understandable so that reasonable inferences may be made about potential vulnerabilities.
- The field is distinct from other, already documented fields. Often fields are repeated in different contexts throughout a protocol. To avoid over-representation of some fields, these duplicates were avoided.

Descriptions and metrics collected from fields in these protocols are listed in Appendix D. Each field was classified as either standard or numeric and the size of the invalid space of the field was recorded. For each field one or more potential vulnerabilities were identified and described along with the size of their vulnerability space.

3.3.2 Standard Field Vulnerability Distribution.

For basic, non-numeric fields, the distribution of vulnerability sizes is essential to the coverage calculation, (29). To estimate this distribution, the vulnerabilities in standard fields in the collected sample were each placed into one of six categories based on their size. A large portion of vulnerabilities covered the entire invalid space, or just one value in the invalid space. Because of this, these two cases were each given their own categories. The remaining vulnerabilities were categorized based on their size relative to the invalid space, $|V|/|I|$. These remaining four categories were:

- $0\% < {}^{|V|}/_{|I|} \leq 25\%$
- $25\% < {}^{|V|}/_{|I|} \leq 50\%$
- $50\% < {}^{|V|}/_{|I|} \leq 75\%$
- $75\% < {}^{|V|}/_{|I|} < 100\%$

This distribution is shown in Figure 14.

Vulnerabilities that occur when any invalid input is applied, that is, vulnerabilities where $|V| = |I|$, were a common occurrence in the protocol analysis. It tended not to matter if the field had many or few invalid values; sometimes it just seemed like any invalid value might cause an issue. In a similar manner, many times one specific value appeared problematic. These two vulnerability sizes were unique in their increased rate of occurrence and independence from the size of the field's invalid space.

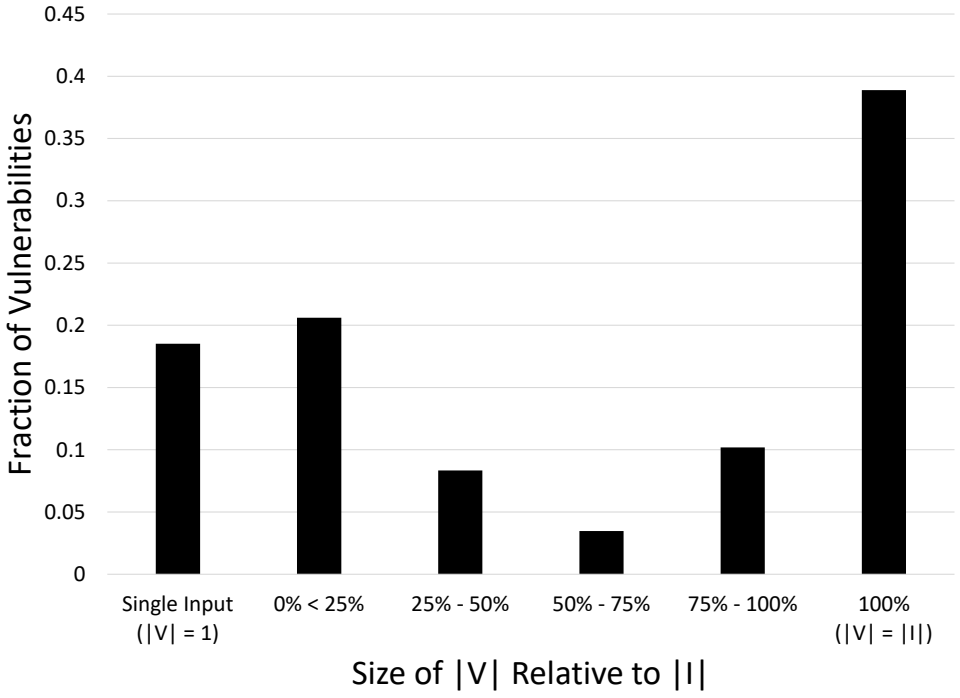

Figure 14. The estimated distribution of vulnerability sizes among non-numeric fields. Data was collected from four open source protocols and is listed in Appendix D.

For these reasons, the model for vulnerability sizes in general assumes that vulnerabilities of size $|V| = 1$ and size $|V| = |I|$ occur at a fixed rate, independent of the value of $|I|$. For the model, these occurrence rates were set at the same levels as observed in the open source protocols. That is single input vulnerabilities, $|V| = 1$, comprise 18.518% of all vulnerabilities and all input vulnerabilities, $|V| = |I|$, comprise 38.889% of all vulnerabilities. This leaves 42.593% of vulnerabilities that have a different size.

To model the distribution of vulnerabilities with sizes between 1 and $|I|$, the four range categories of vulnerability sizes from figure 14 were considered. These appear to suggest that small vulnerabilities, those between 0% and 25% of the invalid space,

make up the largest portion, while larger vulnerabilities, those between 75% and 100% of the invalid space, make up the second largest portion. This observation suggests that the trend of vulnerability size to rate of occurrence is not linear. Rather, a second order polynomial fit was found to apply quite well to these data points. This fit is shown in figure 15.

The distribution of vulnerability sizes can be modeled as a smooth function with the exception of special cases $|V| = 1$ and $|V| = |I|$ by applying this fit. This function can then be sampled and scaled to form a valid PMF for any value of $|I|$. As an example for how this can be done, consider a field with an invalid space of size 5. The probability a vulnerability in this space has a size of less than one, or greater

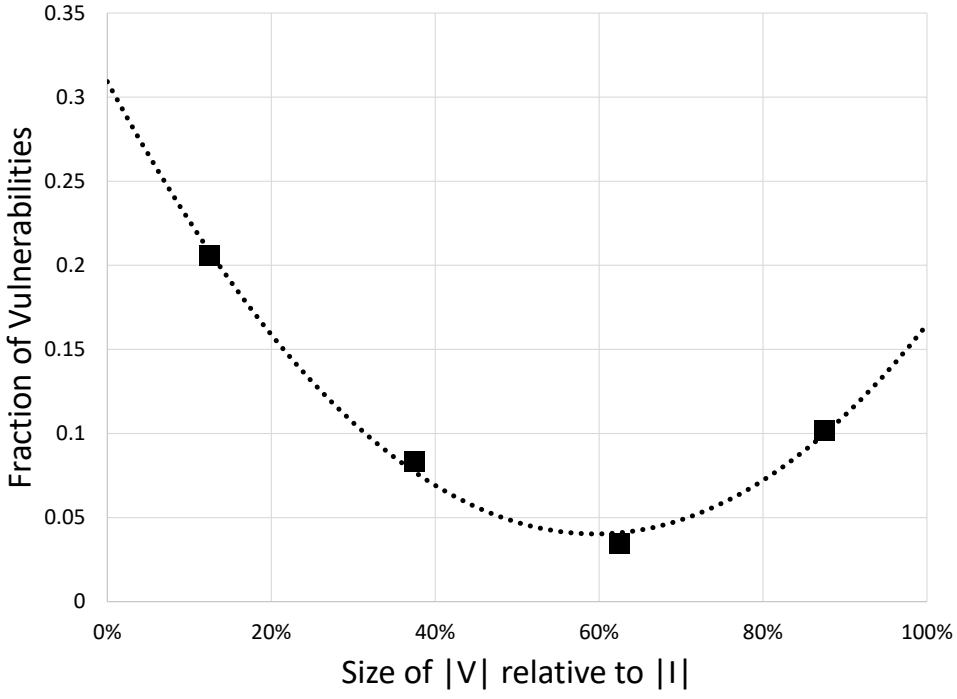

Figure 15. A second degree polynomial least squares fit to the vulnerability size ranges from Figure 14, excluding a size of one and a size of $|I|$

than five is clearly zero.

$$f_X(x) = 0 \text{ when } x < 1 \tag{66}$$

$$f_X(x) = 0 \text{ when } x > 5 \tag{67}$$

The probability a vulnerability is of size 1 or size 5 is fixed per the previous discussion.

$$f_X(1) = 0.18518 \tag{68}$$

$$f_X(5) = 0.38889 \tag{69}$$

The probability a vulnerability is an intermediate size is determined by a sampled and scaled version of the polynomial fit to the reference data. The unscaled equation for the fit is:

$$f_{unscaled}(|V|) = 0.75926 \left(\frac{|V|}{|I|}\right)^2 - 0.90370 \left(\frac{|V|}{|I|}\right) + 0.30920 \tag{70}$$

Applying this to the remaining vulnerability sizes, $|V| = 2$, 3, and 4, unscaled rates of occurrence for these sizes are found:

$$f_{unscaled}(2) = 0.0692 \tag{71}$$

$$f_{unscaled}(3) = 0.0403 \tag{72}$$

$$f_{unscaled}(4) = 0.0722 \tag{73}$$

To form a valid PMF, all values must sum to 1. With the lowest and highest values set, the intermediate values must sum to 0.42593. The scale factor K to make

the middle values sum to this figure can be found using this equation:

$$K = \frac{0.42593}{\sum_{x=2}^{|I|-1} f_{unscaled}(x)} \tag{74}$$

For this example the scale factor works out to be $K = 2.344$. Applying this factor to each unscaled rate of occurrence, the proper PMF value is found. For this example the resulting PMF is:

$$f_X(1) = 0.18518 \tag{75}$$

$$f_X(2) = 0.16224 \tag{76}$$

$$f_X(3) = 0.09451 \tag{77}$$

$$f_X(4) = 0.16918 \tag{78}$$

$$f_X(5) = 0.38889 \tag{79}$$

In general this procedure can be applied to compute the proper PMF when $|I| > 2$ using this formula:

$$f_X(x) = \begin{cases} 0.18518 & x = 1 \\ K\left(0.75926\left(\frac{x}{|I|}\right)^2 - 0.90370\left(\frac{x}{|I|}\right) + 0.30920\right) & 1 < x < |I| \\ 0.38889 & x = |I| \\ 0 & \text{otherwise} \end{cases} \tag{80}$$

The PMF in the special case where $|I| = 1$ is trivially:

$$f_X(x) = \begin{cases} 1 & x = 1 \\ 0 & \text{otherwise} \end{cases} \tag{81}$$

And in the case of $|I| = 2$, $|V| = 1$ and $|V| = |I|$ consume the entire PMF. Since their set values do not sum to 1, they are scaled so that they do sum to 1 while maintaining the same relative proportions.

$$f_X(x) = \begin{cases} 0.32258 & x = 1 \\ 0.67742 & x = 2 \\ 0 & \text{otherwise} \end{cases} \tag{82}$$

3.3.3 Numeric Field Vulnerability Distribution.

Data was also collected on numeric fields. Of the fields selected from the four open source protocols, 28% of them were numeric. This meant there was less information to work with than for the standard fields, but the same general method was applied to this data. The major difference between building the PMF for a numeric field and a standard field is that the numeric field PMF is 2 dimensional. As described in Section 3.1, the numeric PMF has a dimension representing length (analogous to size in standard fields) and a dimension representing position in the invalid space. To build an adequate PMF for numeric fields, the length and position of vulnerabilities in the data set were documented and a PMF was built based on those findings.

The distribution of vulnerability lengths were categorized in the same way as vulnerability sizes were in the standard case. The numeric field distribution of vulnerability lengths is shown in Figure 16.

The distribution of vulnerability position was a bit more difficult to quantify. This was mainly because of a lack of vulnerabilities that did not either start on the lowest value in the invalid space or end on the highest value. It was clear from looking at the vulnerability positions that vulnerabilities tend to either start at the bottom of the range, or end at the top with few residing completely in the middle. This lack of data on middle range vulnerabilities prevented the identification of trends like, for example,

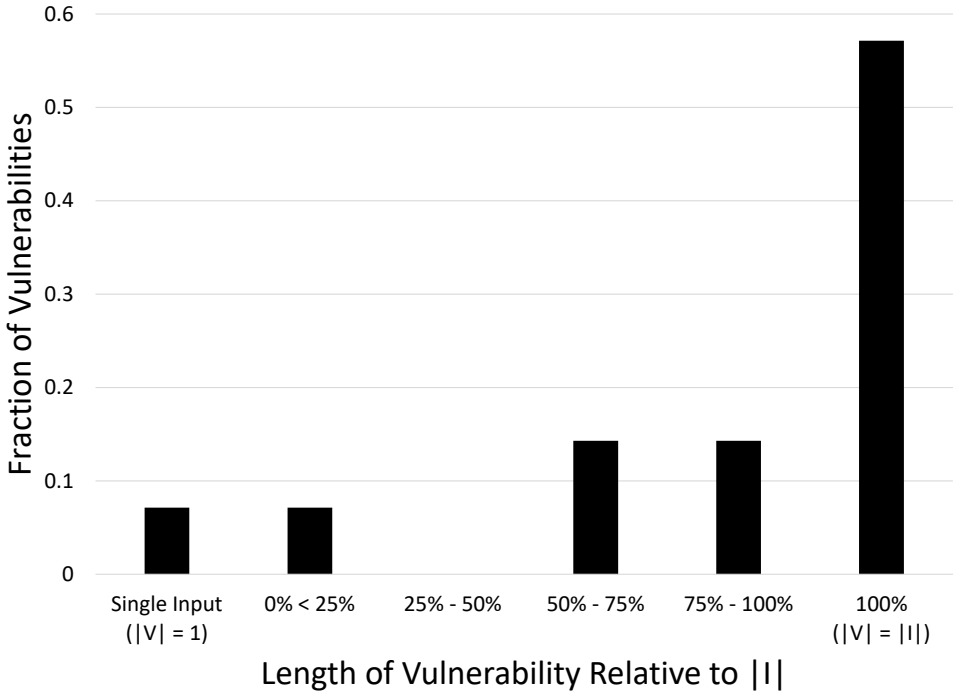

Figure 16. The estimated distribution of vulnerability lengths among numeric fields. Data was collected from four open source protocols and is listed in Appendix D.

"vulnerabilities are more concentrated higher in the range than lower". This led to the lumping all these middle position vulnerabilities into one category along with a category for vulnerabilities that contain the lowest value and vulnerabilities that contain the highest value. Vulnerabilities that cover 100% of the invalid space must include both the lowest and highest values, so these are included in both categories. The vulnerability position distribution is shown in Figure 17.

As was done with the interior categories for vulnerability size in a standard field, a fit was applied to the range categories for numeric field vulnerability length. There appeared no reason to use a polynomial fit again based on the available data, so a linear least squares trend-line was calculated and applied to the four range categories.

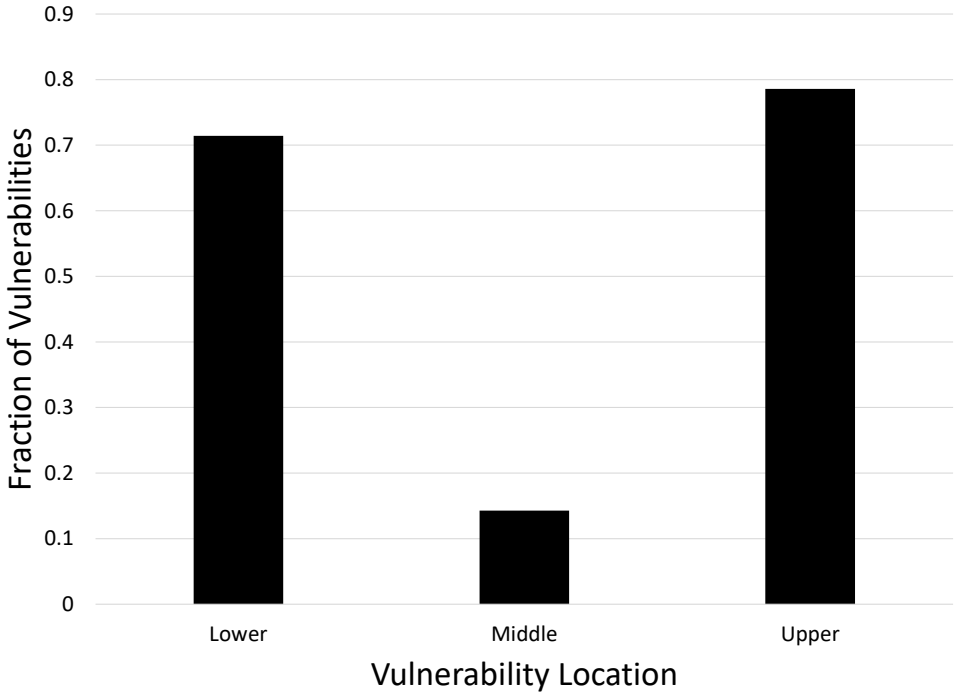

Figure 17. The estimated distribution of vulnerability positions in numeric fields. Lower indicates the vulnerability includes the lowest value in the invalid space, higher indicates the vulnerability includes the greatest value, and middle indicates the vulnerability contains neither extreme value. Data was collected from four open source protocols and is listed in Appendix D.

This fit is shown in Figure 18, and the equation for this line is shown in (83); l is the length of the vulnerability, which is analogous to $|V|$ in standard fields and is introduced in Section 3.1.

$$f_{unscaled}(l) = 0.14286\left(\frac{l}{|I|}\right) + 0.01786 \tag{83}$$

Incorporating the proportions of vulnerabilities with $l = 1$ and $l = |I|$ based on the collected data, a complete, unscaled function, $f_{U1}(l)$ can be arrived at for the distribution of numeric vulnerability lengths:

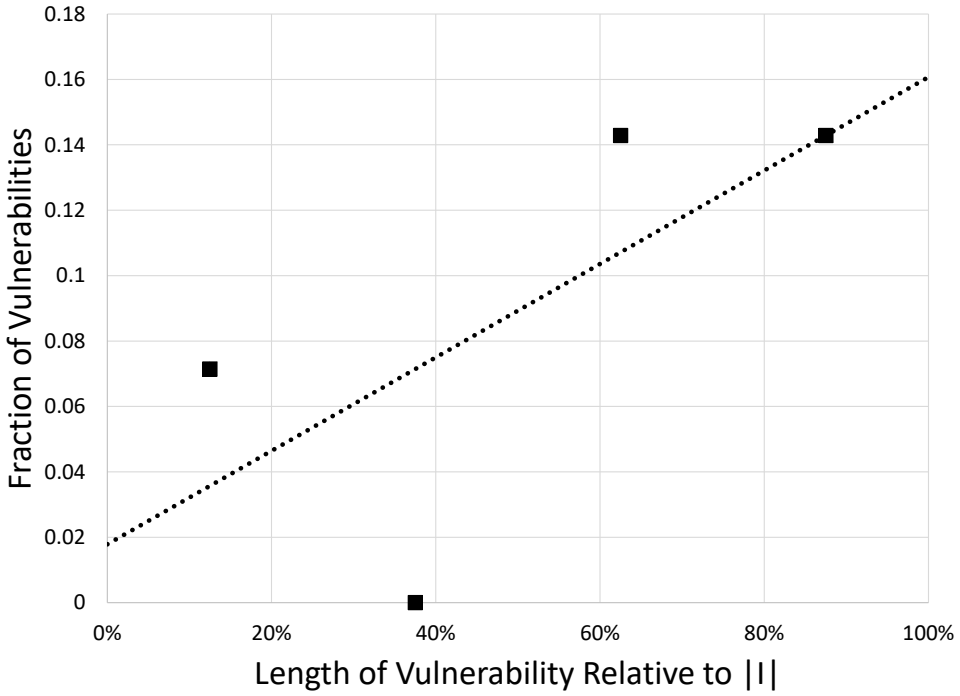

Figure 18. A linear least squares fit to the estimated distribution of vulnerability length ranges in a numeric field from Figure 16, excluding a length of one and a length of $|I|$

$$f_{U1}(l) = \begin{cases} 0.07143 & l = 1 \\ 0.14286\left(\frac{l}{|I|}\right) + 0.01786 & 1 < l < |I| \\ 0.57143 & l = |I| \\ 0 & \text{otherwise} \end{cases} \tag{84}$$

To incorporate position into this function, a modifier, based on the rate of lower, middle, and upper, vulnerability occurrences, is applied. From the data that is shown in Figure 17, the modifier is set as shown in (85). The conditions in the function identify four unique situations. First, if the vulnerability consumes the entire space, then $l = |I|$ and an average of the lower and upper positions is used. Second is a

vulnerability that starts at the lowest value, third is a vulnerability that does not contain the lowest or highest value, and fourth is a vulnerability that contains the highest value.

$$M(v,l) = \begin{cases} 0.75000 & l = |I| \\ 0.71428 & v = 1 \text{ and } l \neq |I| \\ 0.14286 & v > 1 \text{ and } l <= |I| - v \\ 0.78571 & l = |I| - v + 1 \text{ and } l \neq |I| \end{cases} \tag{85}$$

The modifier can then be applied to the length distribution function to reach a two dimensional unscaled distribution function, $f_{U2}(v,l)$ covering both length and position:

$$f_{U2}(v,l) = M(v,l) f_{U1}(l) \tag{86}$$

To properly scale the function to make it a PMF, the scale factor must be found by finding the inverse of the sum across all values of v and l.

$$K = \frac{1}{\sum_{v=1}^{|I|} \sum_{l=1}^{|I|-v+1} f_{U2}(v,l)} \tag{87}$$

The final, valid, PMF for the distribution of vulnerabilities among numeric fields is shown in (88). This PMF is used in the coverage calculator's implementation of (38) and (51).

$$f_X(v,l) = K f_{U2}(v,l) \tag{88}$$

3.3.4 Other Special Field Types.

Ideally, for the most accurate coverage calculation possible, these PMFs would account for everything a tester knows about the nature of vulnerabilities. One addi-

tional way these PMFs can be improved is by developing seperate functions for special field types that the tester may have some insight into, or that may have an obviously different distribution of vulnerabilities than the two previously described. Due to time constraints and a limited number of fields analyzed from open source protocols, these special PMFs were not pursued for this thesis. Developing these special PMFs may improve the quality of the coverage calculator, so it should be considered for future work.

Some specific field types that likely have real vulnerability distributions that differ from the standard ones estimated earlier in this section include:

- **Repeating Relation Fields** — These fields indicate how many blocks, fields, bits or bytes follow in a message. Section 3.1.3.1 describes how the invalid space of this field is treated as two dimensional since there are two elements that determine this fields validity: its value, and the actual number of blocks, fields, bits or bytes that follow. The general estimated distribution of vulnerability sizes probably underestimates the size of these vulnerabilities since it is unlikely that a single combination of value and repeats would trigger a vulnerability. For example, a case where the field says 2 bytes should follow and 7 actually follow is unlikely to trigger a vulnerability if a value of 2 with 8 bytes actually following does not trigger one.

- **Choice Tokens and Relations** — These fields indicate how other bits in a message should be interpreted. An invalid value in this field often mean that the receiver will not be able to interpret some bits. Again the general estimated distribution of vulnerability sizes probably underestimates the typical size of these vulnerabilities. If a vulnerability is triggered because a receiver cannot process some bits, it would be expected that any invalid value in this field would trigger this vulnerability.

- **Second and Higher order Field Combinations** — In this thesis, the general PMF is applied to field combination coverage calculations as well as field coverage calculations. Certainly the distribution of vulnerability sizes in field combinations is vastly different than with fields. The field combination invalid spaces tend to be orders of magnitude larger than the invalid field spaces. Vulnerabilities in these spaces probably rarely cover the entire space. They are likely confined to regions where one field is invalid, and a vulnerability is triggered based on a valid value in a second field, or they may be confined to regions where both fields are invalid. For this reason, the general PMF would be expected to overestimate the vulnerability sizes in this case. It also should be pointed out that the coverage calculator developed in this thesis does not handle numeric fields in a field combination any differently than basic fields. If the time were taken to model these higher order fields with thier own PMFs, combinations that are part or all numeric can be treated properly.

3.4 Coverage Calculation Procedure

The previous sections in this chapter developed the equations and methods necessary to compute ExCov. This section takes these equations and methods and turns them into a procedure to reach a final coverage value $\mathbb{C}_{FuzzSet}$. The procedure assumes that the tester has a set of test cases and a model for the protocol being fuzzed.

Set-up

1. Decide which final coverage approximation value, $\mathbb{C}_{FuzzSet}$, is to be computed. Section 3.2.4 discusses three different approaches and lists their requisite equations: (63), (64), and (65).

2. Based on the selection in step one, find the order coverage values, $\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_{R_{Max}}$

that need to be calculated. For example, if $\mathbb{C}^1_{FuzzSet}$ (shown in (64) on page 80), is chosen, $\mathbb{C}_1, \mathbb{C}_2$, and $\mathbb{C}_3$ all need to be calculated and $R_{Max} = 3$.

3. Identify the value of each field present in each test case.

4. Of those values identified in step 3, identify which values are invalid. Values that are invalid lie in their field's invalid space.

5. Identify all sets of fields that are Invalid in Combination (IIC).

First Order Coverage Calculation

6. If $\mathbb{C}_1$ was identified in step 2, proceed to step 7. Otherwise proceed to step 13.

7. Apply the first order test case limitation in Section 3.2.1 on page 65. This means that all test cases which have zero or two or more invalid fields as found in step 4, or have an IIC set as identified in step 5, are no longer considered in the first order coverage calculation.

8. Apply the first order field limitation in Section 3.2.1 on page 65. This means that only fields with an invalid space are considered for the rest of the first order coverage calculation.

   For steps 9 to 12, the steps shown in Section 3.2.1 on page 65 are followed.

9. For each test case not eliminated in step 7, identify the one field that is invalid.

10. Sort the test cases into groups based on the fields identified in step 9.

11. For each group of test cases, calculate first order field coverage, $\mathbb{C}_1$, of the invalid field using the appropriate equation. To find the appropriate equation, reference Table 2 which is reproduced here for convenience:

Is the field numeric?	Does the field contain weighted values?	Then use eq.	on page
no	no	(29)	47
yes	no	(38)	55
no	yes	(50)	59
yes	yes	(52)	60

12. Combine field coverage values to find $\mathbb{C}_1$ using (57) if any field identified in step 8 is weighted or by using (58) if no field identified in step 8 is weighted. Note that for each field identified in step 8, $E[N_i] = 0$ for (57).

Multi-Order Coverage Calculation

13. Let the current order R be 2.

14. If $\mathbb{C}_R$ was identified in step 2, proceed to step 15. If all coverage order values identified in step 2 have been found, proceed to step 22. Otherwise proceed to step 21.

15. Apply the R^{th} order test case limitation in Section 3.2.2 on page 71. This means that test cases with no invalid values or IIC sets are not considered for R^{th} order coverage. Also any test case where the number of fields with an invalid value and plus number of fields present in all IIC value sets is greater than R are not considered for R^{th} order coverage.

16. Apply the R^{th} order field combination limitation in Section 3.2.2 on page 72. This means that any field combination with no invalid space is not considered for R^{th} order coverage.

    For steps 17 to 20, the steps shown in Section 3.2.2 on page 72 are followed.

17. For each test case not eliminated in step 15, identify all invalid fields and IIC value sets.

18. For each combination of R fields, identify all test cases that have invalid and IIC values entirely contained in the fields of the combination.

19. For each field combination of R fields, compute the coverage based on the identified value combinations in step 18. Use equation (29) on page 47 for these multi-order spaces.

20. Combine field combination coverage values to find $\mathbb{C}_R$ using (62) if any field combination identified in step 16 is weighted or by using (61) if no field combination identified in step 16 is weighted. Note that for each field combination identified in step 16, $E[N_i] = 0$ in both equations.

21. Add one to R to advance to the next order. Go to step 14.

Combining for a Final Result

22. Compute $\mathbb{C}_{FuzzSet}$ using the method identified in step 1 and the values from steps 12 and 20.

IV. Implementing the Criterion

To demonstrate that the approach presented in Chapter III can be successfully implemented and used to improve fuzz test coverage, a model for protocol specifications, a coverage calculator, and a high coverage fuzz generator were developed. These three tools were combined into one C++ visual studio project called ExFuzz.

The name ExFuzz, short for Expected Fuzz or Expected Fuzzer, was chosen as it conveys two important characteristics of the coverage criterion on which the tool is based. First, as can be seen throughout Chapter III, the metric relies heavily on the *expected* value operator to account for uncertainty about the type and location of vulnerabilities in a protocol implementation. Second, the criterion is designed to output a coverage percentage that means what a tester *expects* it should mean. This clarity allows the tester to use the metric effectively without knowledge of precisely how it is calculated.

ExFuzz is comprised of three tools: DataModel, ExCov, and GenFuzz. Each tool produces a C++ object that can be queried by the main application, ExFuzz.

- **DataModel:** Creates a data model object from an Peach Pit XML file with the extensions described in Section 4.1.2. The DataModel tool is described in Section 4.1.3.
- **ExCov:** Computes the coverage of a set of test cases based on the criterion presented in Chapter III. This tool is described in Section 4.2.
- **GenFuzz:** Creates a set of fuzzed test cases from a data model object. The test cases are designed to achieve high coverage per the ExCov criterion. This tool is described in Section 4.3.

4.1 Creating a Data Model

A protocol specification is a document that standardizes many aspects of a protocol. These specifications can cover anything from the electrical characteristics of the signals to medium control and fragmentation. For example, the Internet Protocol version 4 (IPv4) standard covers addressing conventions, fragmentation procedures and IP header format [3], while Military Standard 1553B (MIL-STD-1553B) covers the electrical characteristics of the bus, medium access control and word formats [2].

Many of a protocol's characteristics described by such standards can be "fuzzed" in some way. That is, the voltages on a 1553 bus could be manipulated beyond the allowable levels to attempt to trigger an adverse reaction from a SUT. But this is a departure from the understood definition of a fuzz test. For this thesis, a fuzz test attempts to expose a vulnerability in the software of the SUT, not the hardware.

Another characteristic of a protocol that could be fuzzed is medium access control. the Ethernet standard, 802.3-2015, for instances specifies how to share a medium using Carrier Sense Multiple Access with Collision Detection. Aspects of this medium access protocol could be fuzzed, bending the rules defined in the standard slightly to cause an adverse affect on the SUT [6]. This type of fuzzing will also not be considered as the opportunities and challenges associated with it are very different from the type of fuzzing considered in this thesis.

This thesis considers the sections in protocol specifications that assign meaning to sets of bits, or fields. For example, in the case of the IPv4 protocol, this would be section 3.1, *Internet Header Format*, which describes the fields of the IPv4 header and gives them meaning. Likewise for the MIL-STD-1553B standard and the Ethernet standard, the sections that describe the meaning of the bits in the headers of those messages are the target of the fuzzing described in this thesis. This fuzzing involves sending messages that violate the specification's limitations on the values of

the protocol's fields.

Protocol specifications are not written in any standard way that would allow a software fuzzing tool to automatically and easily interpret them. Rather, the specifications rely on humans to read them and build systems or tools that follow the rules as described. In fact, many bit maps, graphical representations of the meanings of sets of bits in a message, are drawn using standard text characters. This is true in MIL-STD-1553B and the IPv4 standard. In order for a software tool to understand the rules governing an instantiation of a protocol, the protocol specifications must be converted into a common form that can be understood by a software application.

Ideally this form is human and machine readable, and is specific enough to completely specify all the relevant rules from a protocol specification yet general enough to accommodate many protocols. For the type of fuzzing considered in this thesis, the model must cover all possible realizations of a message, but need not cover the order or interactions between multiple messages. Peach provides an excellent starting point to crafting such a model.

4.1.1 Peach Pit Modeling.

As discussed in Chapter II, Peach is a widely used fuzzing platform that has the capability to fuzz network protocols. A fundamental element of the Peach platform is the Peach Pit which is an XML document that defines "the structure, type information, and relationships in the data to be fuzzed" [10]. A Peach Pit contains three main parts, the Data Model, the State Model and the Agent.

The Agent describes the monitoring system for the SUT and is especially useful if Peach can be installed on the SUT. For military data link fuzzing, a Peach defined agent is not relevant as discussed in Sections 1.4 and 2.1.2.2. The State Model describes when message transmissions and receptions are allowed to occur. This section

can be used to model the aforementioned interactions between messages or rules involving medium access control. Since the scope of this thesis is limited to fuzzing within a message, the State Model is not of use.

The Peach Pit Data Model describes the composition of a message and defines the relationships between elements within a message. This data model is an excellent place to start in the effort to create a generic model of a message defined by a protocol specification. As a simple example, consider the MIL-STD-1553B command word definition shown in Figure 19 (retrieved from [2], page 6).

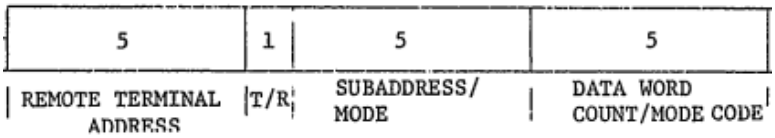

Figure 19. Message standard for a MIL-STD-1553B command word

This 16 bit message can be represented as an XML data model. This is shown in Figure 20. The Peach data model represents messages as a collection of fields. The order, existence, and values of these fields are determined by other elements in the model. The elements used to describe fields are *Number* and *String*. *String* may be useful when the data model is for a file or ASCII based protocol, but generally in military data links each field is best represented as a binary number using the *Number* element. Each element can be further described using attributes. The attribute *size*

```
<DataModel name="Command_Word">
  <Number name="Remote_Terminal_Address" size="5"/>
  <Number name="Transmit/Receive" size="1"/>
  <Number name="Subaddress/Mode" size="5"/>
  <Number name="Data_Word_Count/Mode_Code" size="5"/>
</DataModel>
```

Figure 20. MIL-STD-1553B command word represented in a Peach data model

is required for the element *Number*; it sets the number of bits in the field. Other useful elements from the Peach data model language include:

1. **Block:** Blocks are used to group a set of like fields together. Blocks can be repeated allowing the same fields to appear multiple times in a message.

2. **Choice:** Choices are used to accommodate protocols where the value in one field controls what fields appear elsewhere in the message. A simple example of this would be the version field of the IP header. If this field has a value of 4, the subsequent fields are described by the IPv4 standard. If it has a value of 6, the subsequent fields are different and are described by the IPv6 standard. Military data link protocols often have this quirk and require the choice element.

3. **Relation:** The relation element is a child of a field element and describes how the value in one field is related to the value in another field, block, or choice. For example, Peach uses the relation element to model checksums. This element can also be used with fields like the Data Word Count field shown in Figure 19 by relating the value in the parent field to the number of occurrences of a different field, block or choice.

4. **Hint:** The hint element is also a child of a field element. It is used to relay more information about the field such as which values are valid and which are invalid.

Open source protocols like those discussed thus far are too simple to illustrate the necessity of these other data model elements. Unfortunately the standards of the protocols that this thesis targets, military data link protocols, can not be reproduced in this setting. For this reason, fictitious protocols were created that mimic the features of actual military data link protocols. These protocols can be described and

modeled freely in this setting. Appendix B describes these protocols and shows how they can be modeled using a Peach Pit XML file with some extensions.

4.1.2 Custom Peach Pit Extensions.

The Peach data model has a number of limitations that prevented its exclusive use in this thesis. To overcome these limitations, some additions were made to the data model that allow more information about each field to be conveyed to a fuzzer or coverage calculator through an XML file. Other changes were also made that allow some structural quirks of military data link protocols to be accommodated. The extensions made were:

1. **Valid and Invalid Values:** Attributes to the hint element were added that allow for specification of all valid and invalid values of a field. Peach had already had a valid values attribute for the hint, but it was only applicable to one peach mutator and the syntax was inconvenient for specifying large ranges of values. This modification allows for a complete listing of valid and invalid values in more compact notation. Specifically, a range of values is represented, throughout the data model, as a list of values and ranges separated by commas. for example, the values of 1, 2, 3, 4, 7, 8, 9, 11, 15, and 16 can be compressed to the text string "1-4,7-9,11,15,16".

2. **Field Type:** The coverage criterion ExCov categorizes fields as either standard or numeric based on the type of data they contained. Section 3.1.3.2 discusses the differences between these two types. To separate them in the data model, a *type* attribute was added to the hint element that can be either "Standard" or "Numeric".

3. **Rule Set:** A unique situation arises when a choice element is located within a

repeating block element. In this case, the choice element may appear multiple times in a single message. Protocols often place rules on which choice can be made on which repeat. The Restaurant Locator Protocol, described in Section B.3 provides an example of this. The coverage criterion handles this case as a type of field combination as described in Section 3.2.3.3. To facilitate automatic detection of rule violations, the set of rules must be included in the data model.

A *RuleSet* element was therefore added that appears as the immediate child of a repeating block containing a choice. The rule set has one child element for each rule placed on the choices. Three types of rules were found to be necessary to cover the situations seen in military data link protocols. Support for these three rules was added to ExFuzz; they are: 1) a position rule: for example, choice 1 must be made during the first repeat. 2) a repeat rule: for example, choice 2 must be made no more than three times. And 3) a sequence rule: for example, choice 4 must always be made the repeat after choice 3. Figure 21 shows the rule set used to describe the word order rules of the Restaurant Locator Protocol.

```
<RuleSet block="OuterRepeatingBlock" choice="Word_Choice">
  <Position token="0" present="yes" position="1" />
  <Repeats token="0" precision="exactly" times="1" />
  <Repeats token="1" precision="no more than" times="1" />
  <Repeats token="2" precision="no more than" times="1" />
  <Repeats token="3" precision="no more than" times="3" />
  <Sequence token="1" token2="2" present ="yes" bora="after" proximity="immediately" />
  <Sequence token="2" token2="1" present ="yes" bora="before" proximity="immediately" />
</RuleSet>
```

Figure 21. An example of the rule set extension to a Peach Pit data model. This example is from the Restaurant Locator Protocol described in Section B.3.

4. **Token as a Relation:** The Peach Pit data model uses the token attribute of a field element to convey that the value in that field determines what choice was made. This proved to be inadequate for representing some military data link protocols. Sometimes the field that determined which choice was to be made

was in a different part of the message and could not act as a token field. For this reason a new type of relation element called a *choice relation* was added. To select this type of relation, the *type* attribute of a relation element is set to "choice". This allows these fields to act as token fields without being within associated choice block as the Peach data model required.

5. **Repeating Relation Adjustment:** The Peach Pit data model allowed for a type of relation element that linked the value in a field to the number of repeats of a block. However, the value in the field had to be the exact number of repeats of the block. In some military data link protocols, the number of repeats was related to, but not precisely, the value in the related field. For this reason, an *adjustment* attribute was added to the relation element. The value of this element was the difference between the value in the field and the actual number of block repeats that value represented. In the example of the Restaurant Locator Protocol, the *Num_Words* field has an adjustment of +1 meaning that when the field has a value of 2, 3 words actually occur in the message.

6. **Field Weighting:** The data model also needed a mechanism to convey information about which fields are more likely to contain vulnerabilities than others. This information is incorporated into the coverage criterion in Section 3.2.1, specifically on page 67 and in (57). The value is termed field weight, and in the data model is an attribute, *weight*, that belongs to a hint element. It can take on any numeric value.

7. **Value Weighting:** In a similar manner to field weighting, the data model required a mechanism to convey the weighting of individual values within a field. Section 3.1.3.3 is devoted to the incorporation of this type of information.

To convey value weights a new element was created called *Weighted*, as a child of a field element. The weighted element was itself given a child element, *Range* that can repeat within a weighted element block any number of times. Each range element assigns one weight value, through the *weight* attribute, to a range of values for the field, using the *values* attribute. The same range notation is used as when describing the valid and invalid space.

4.1.3 DataModel Tool.

The DataModel tool takes a Peach Pit with the custom extensions described in Section 4.1.2 that models a protocol, and builds a C++ object. Figure 22 shows this process.

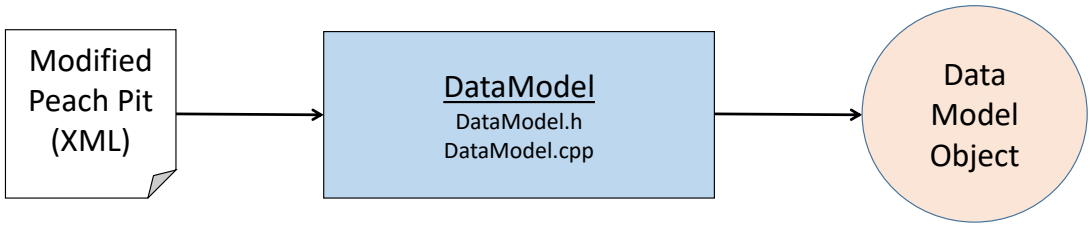

Figure 22. Inputs and Outputs associated with the DataModel tool

The DataModel tool uses an open source XML parser called Tiny XML 2 [24] to read in the XML file. It then builds a C++ object that is designed to easily interact with the other fuzzing tools. The object contains a number of different custom data structures called nodes.

These nodes are related to some of the elements in a Peach pit. For example, the C++ object defines a field node, a block node, and a choice node. Each node has parameters unique to its type. For example, the field node has a parameter for valid values, which contains the information presented in the *valid* and *invalid* attributes of the *hint* element. The block node contains parameters called *minOccurs*

and *maxOccurs*, to carry the values supplied by the attributes with the same names in the Peach Pit *Block* element.

Each node also has at least one child and one parent node. This allows the nodes to be conceptually connected by edges in a graph. To build or interpret a message using the data model, start at the first node, then proceed to its child node, then to that node's child, and so forth. Along the way, every field node encountered represents a set of bits in the message and all the properties associated with those bits. Traversing the data model in this way not only places the fields in the proper order, but allows for sets of fields to repeat, and for choices to be made that include some fields and exclude others.

Any message for a given protocol can be created by tracing a path through this node map, creating fields as their nodes are encountered. Similarly any message can be parsed by following the node map, interpreting bits as their fields are encountered, and making choice and repeat decisions based on their nodes parameters, and special fields in the message. All the information necessary for creating and parsing messages is contained in the data model object.

Figure 23 shows the node map for the Restaurant Locator Protocol which is described in Appendix B. This map does not convey any information about the parameters within each node, but simply shows how they are arranged.

4.2 Building a Coverage Calculator

The Coverage Calculator tool, ExCov, takes a data model object, and a set of test cases in the form of a text file and returns a value between 0% and 100%. Figure 24 shows this process.

ExCov looks at the data model object to find invalid spaces and characteristics of each field in the protocol. It then applies the field, and field combination limitations

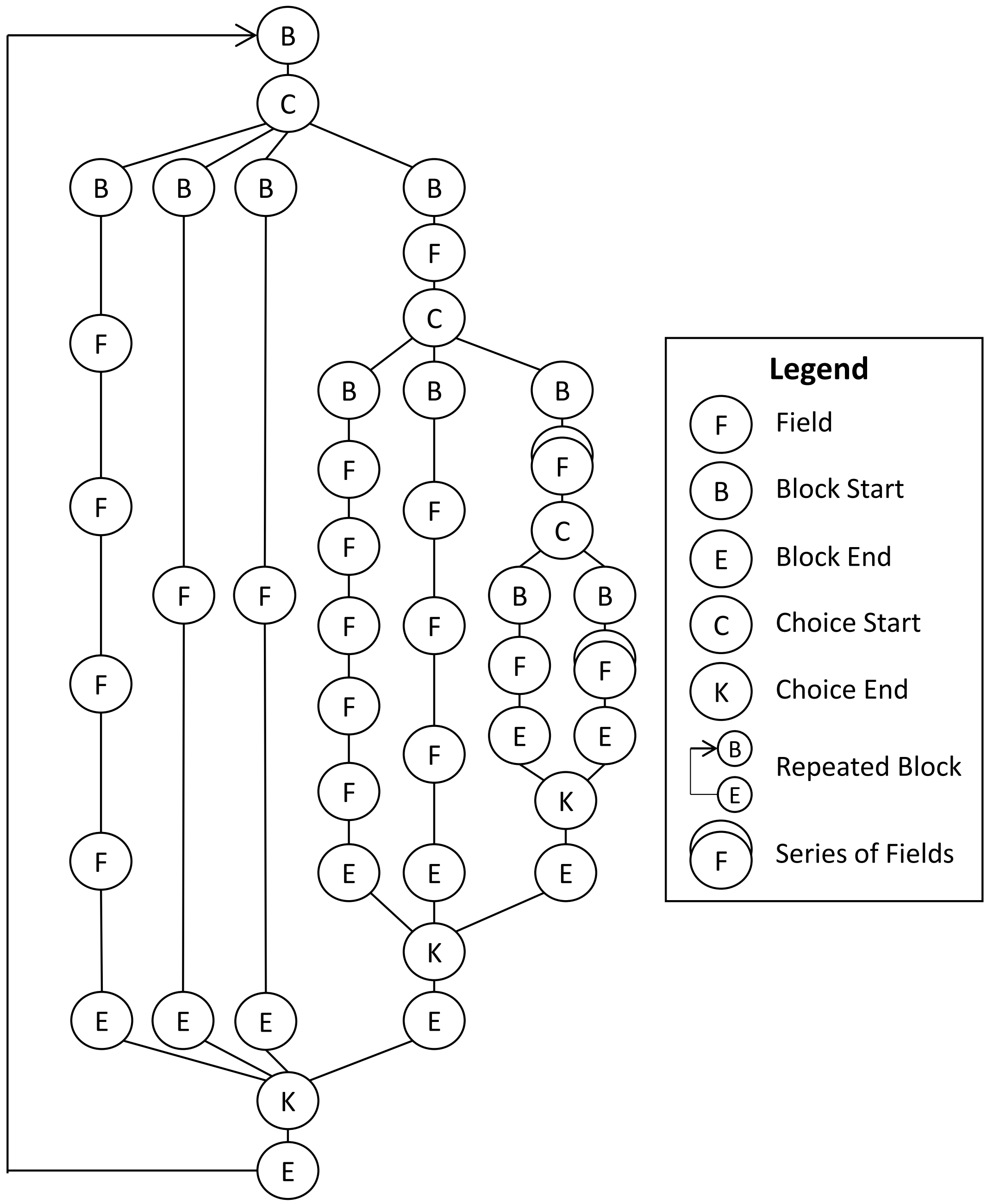

Figure 23. Node Map for the Restaurant Locator Protocol

(described in Sections 3.2.1 and 3.2.2.2 respectively), to eliminate some fields and field combinations from the calculation.

The input test cases text file contains one test case per line in hexadecimal. ExCov

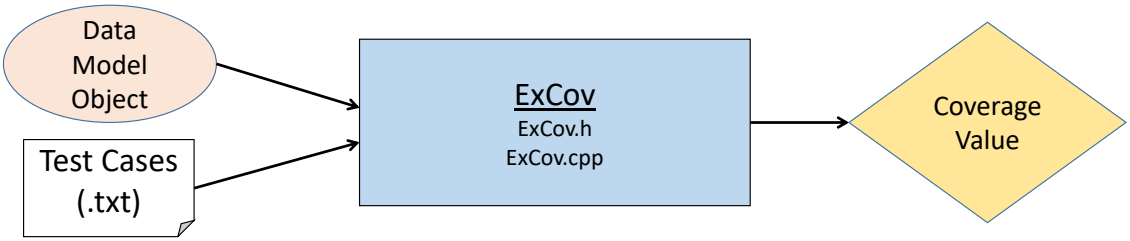

Figure 24. Inputs and Outputs associated with the ExCov tool

parses each test case by converting them to binary and then comparing them to the data model object. Once the tool understands a test cases's value for every field, it applies the test case limitations (described in Sections 3.2.1 and 3.2.2.2) to remove test cases that contribute no coverage.

The tool keeps track of which invalid values have been tested as it parses test cases, and then applies the appropriate equations to compute coverage for each order: (29), (38), (50), (52), (57), (58), (61), or (62) depending on the type of field and the information available. ExCov uses the simplified coverage value, $\mathbb{C}^2_{Fuzz\ Set}$, as its final value. This is computed using (65).

While ExCov implements almost everything described in Chapter III, there is still work that can be done to improve the calculator. Currently ExCov does not support:

- Invalid in combination (IIC) pairs of fields.
- Numeric fields in the multi-order case
- Unique vulnerability distribution functions, $f_X(x)$, for field combinations, choice relation fields, token fields, or repeating relation fields.
- Field combination value weights
- Field combination weights

No significant barriers to the implementation of these functions in ExCov are foreseen. Developing unique vulnerability distribution functions would require the

collection of significantly more protocol data than was collected for this thesis (shown in Appendix D). Field combination weighting, field combination value weighting and IIC pairs will require additional extensions to the Peach Pit XML data model, but these should not be difficult to make. These functions were not incorporated into the tool but are left as future work.

4.3 Building a Generative Fuzzer

The GenFuzz tool takes a data model object and produces a set of fuzz test cases. Figure 25 shows this process.

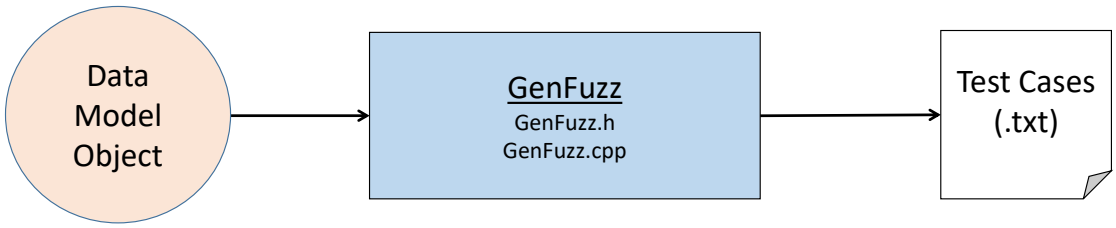

Figure 25. Inputs and Outputs associated with the GenFuzz tool

The tool is designed to provide high coverage based on the ExCov criterion. To do this, it focuses on first order coverage. The method creates test cases in which every field is valid except one. The invalid value changes fields from test case to test case so that each field with an invalid space is fuzzed at least once before any are fuzzed twice. No invalid values repeat until the test case has achieved 100% first order coverage. The valid values that make up the rest of each message are selected randomly based on their field's valid space.

To ensure that every invalid value is tested without repeating any, GenFuzz employs a simple algorithm. For each field it first finds the size of the invalid space, $|I|$, from the information in the data model object. GenFuzz then generates invalid values to test in such a way that none get repeated, and the values are well spaced

within the invalid space.

To do this, the algorithm finds a value this thesis terms as the "jump". The jump, call it j, is an integer such that $j < |I|$ and $GCD(j, |I|) = 1$ where GCD stands for greatest common divisor. Since j and $|I|$ have no common factors, the following can be said: *No multiple of j, call this kj, is evenly divisible by $|I|$ if $k < |I|$*. This means that for all $k < |I|$, $kj \ \% \ |I| \neq 0$, where % is the modulus operator. Further, the set $\{kj \ \% \ |I|; \ 0 \leq k < |I|\}$ is the complete set of positive integers less than $|I|$. This is illustrated in Table 4 with $|I| = 10$.

Table 4. Listing of $k \cdot j \ \% \ 10$ result for all $0 \leq k < 10$ and all $j < 10$ and $GCD(j, 10) = 1$.

k	$j = 1$	$j = 3$	$j = 7$	$j = 9$
0	0	0	0	0
1	1	3	7	9
2	2	6	4	8
3	3	9	1	7
4	4	2	8	6
5	5	5	5	5
6	6	8	2	4
7	7	1	9	3
8	8	4	6	2
9	9	7	3	1

A few key observations can be made by looking at the example in Table 4. First, each column contains all the values between zero and nine without repeats. Second, the central columns have somewhat scrambled the order of these values. In fuzzing it may be undesirable to test a field's invalid values sequentially, it would be more effective to test them by jumping around in the space. Consider the case of numeric fields presented in Section 3.1.3.2. This simple modulus based algorithm is therefore the basis for generating invalid values in GenFuzz.

To implement this algorithm in practice, a random non-negative integer less than $|I|$ is selected. This value is the index of the first invalid test case that will be tried.

For example, let $I = \{6, 7, 8, 10, 12\}$, so that $|I| = 5$. Let the randomly selected integer be 2. Starting with an index of 0, the invalid value of 8 has an index of 2, and thus 8 will be the first value fuzzed. If the jump was calculated to be 3, the index of the next value to be tested would be $(2 + 3)$ % 5, or 0. Thus 6 would be the next value tested. This continues until the algorithm returns that 8 is to be fuzzed again, in which case every value will have been fuzzed.

Through experimentation, it was found that jump values somewhat near ½ of $|I|$ provided a set of well spaced values. Based on this observation, two simple equations were used to select a jump. For invalid spaces larger than 10, the tool computes, using integer division, a jump of:

$$j = \frac{|I|}{2} + \frac{|I|}{7} - 1 \tag{89}$$

And for spaces 10 or smaller:

$$j = \frac{|I|}{2} - 1 \tag{90}$$

The method will only work if $GCD(j, |I|) = 1$ however, so the $GCD(j, |I|)$ is found using the euclidean algorithm [1]. If it is not 1, j is increased by one and the tool again finds the GCD. This continues until a valid jump is found.

GenFuzz takes the invalid fuzzed value and all the random valid values and builds a hexadecimal message based on the data model. It prints this message to a file, and then moves on to the next field to be fuzzed. Since the output of GenFuzz and the text file input to ExCov have the same format, the coverage provided by the test set GenFuzz generates can be immediately found by running ExCov.

Figure 26 shows the first order coverage result for 30 test cases generated by GenFuzz. Notice that fields 12 and up all have an invalid space which has a size of

```
Computing First Order Coverage...

Field 0 (T) has 4 of 4 values tested. That is a coverage of 100%   Weight: 10
Field 1 (R) has 4 of 66 values tested. That is a coverage of 69.336%
Field 4 has 3 of 93 values tested. That is a coverage of 65.9109%
Numeric field 6 has 3 of 910 values tested with a coverage of 63.0819%
Field 10 (T) has 3 of 5 values tested. That is a coverage of 90.9702%
Field 12 has 1 of 1 values tested. That is a coverage of 100%
Field 13 has 2 of 2 values tested. That is a coverage of 100%
Field 14 has 3 of 3 values tested. That is a coverage of 100%
Field 16 has 1 of 1 values tested. That is a coverage of 100%
Field 17 has 3 of 3 values tested. That is a coverage of 100%
Field 20 has 1 of 1 values tested. That is a coverage of 100%
Field 22 has 1 of 1 values tested. That is a coverage of 100%
Field 23 has 1 of 1 values tested. That is a coverage of 100%

First Order Coverage: 94.9681%
```

Figure 26. Partial output of ExCov with 30 test cases generated by GenFuzz for the Restaurant Locator Protocol

three or less. These fields were completely fuzzed because in the first 30 test cases, GenFuzz was able to fuzz each field at least three times if it needed to be. The first two fuzzable fields each have four values tested; this shows that GenFuzz got to these fields a fourth time. The next field, field 4, has only three values tested because GenFuzz had not reached it a fourth time. If GenFuzz had been allowed to continue for a 31_{st} test case, this field would have been tested a fourth time, and coverage would have increased.

Figure 27 shows the first order coverage result for 1200 cases generated by GenFuzz. That is enough test cases to completely cover all first order spaces, and thus first order coverage is 100%. GenFuzz reaches 100% first order coverage in the fewest number of test cases possible since it does not repeat invalid values.

```
Computing First Order Coverage...

Field 0 (T) has 4 of 4 values tested. That is a coverage of 100%   Weight: 10
Field 1 (R) has 66 of 66 values tested. That is a coverage of 100%
Field 4 has 93 of 93 values tested. That is a coverage of 100%
Numeric field 6 has 910 of 910 values tested with a coverage of 100%
Field 10 (T) has 5 of 5 values tested. That is a coverage of 100%
Field 12 has 1 of 1 values tested. That is a coverage of 100%
Field 13 has 2 of 2 values tested. That is a coverage of 100%
Field 14 has 3 of 3 values tested. That is a coverage of 100%
Field 16 has 1 of 1 values tested. That is a coverage of 100%
Field 17 has 3 of 3 values tested. That is a coverage of 100%
Field 20 has 1 of 1 values tested. That is a coverage of 100%
Field 22 has 1 of 1 values tested. That is a coverage of 100%
Field 23 has 1 of 1 values tested. That is a coverage of 100%

First Order Coverage: 100%
```

Figure 27. Partial output of ExCov with 1200 test cases generated by GenFuzz for the Restaurant Locator Protocol

V. Results and Discussion

In this chapter, fuzz test sets are created manually, by mutating valid messages, and by using GenFuzz. These test sets are compared against each other using ExCov, specifically the $\mathbb{C}^2_{Fuzz\ Set}$ metric presented in (65), to answer some questions about the best approach to fuzzing. Since all the tools in ExFuzz are used in these comparisons, this chapter serves as validation that the tools work, and, in the case of the ExCov tool, that the proposed coverage criterion is implementable.

The three protocols created for this thesis and described in Appendix B are used as the basis for these results and comparisons. Since the Restaurant Locator Protocol is designed to be similar to military data link protocols, results regarding this protocol may be directly useful to testers today.

5.1 Mutative Fuzzing Methods

The fuzzing framework presented in Section 1.4 describes the process of mutative fuzzing as a mutative fuzzer operating on a protocol recording. Since the protocols used in this section are not implemented anywhere, valid messages were generated that can act as protocol recordings.

To generate these valid messages, spreadsheets were created in Microsoft Excel that built hexadecimal messages with randomly selected valid fields. These messages were then converted into their binary form and, in a second spreadsheet, bit mutations were applied with a probability entered by the user based on the methods described in this section. ExCov was then used to compute the coverage of the test sets, returning coverage values that are used in analyses later in this chapter.

Two main approaches to mutative fuzz set generation will be discussed here. First is the simple case of sending fully randomized bits. Second is the more complex case

of sending messages with only a few bit flips.

5.1.1 Fully Random Bits.

To generate a fully randomized message, the probability of a bit flip was set to 0.5 in the mutative fuzz generation spreadsheets. In general, random fuzzing is not very effective based on the ExCov criterion. The reason for this is straightforward. Because of the test case limitation (Section 3.2.1, page 65) a message only provides first order coverage if it has one and only one invalid field. Any message that has two or more invalid fields provides no first order coverage, and any message that has three or more invalid fields provides no second order coverage.

For long messages with lots of fields with large invalid spaces, many fully random messages will have three or more invalid fields and provide no coverage whatsoever. This method is more effective if a message has less fields with smaller invalid spaces. Table 5 shows how each of the created protocols is covered by 1000 random test cases along with the number of fields, and total invalid space in the protocol (that is, $|I|$ summed for all fields). Random Fuzzing works best for the Meal Protocol since it has a much smaller total invalid space than the other protocols. It also works fairly well for Simple Protocol compared to Restaurant Locator Protocol largely because Simple Protocol has only 3 fields.

Table 5. A comparison of the coverage provided by the fully random mutative fuzzing approach with 1000 test cases.

Protocol	Number of Fields with an Invalid Space	Total Invalid Space (all fields)	Coverage (1000 Test Cases)
Simple Protocol	3	690	69.4%
Meal Protocol	8	330	77.9%
Restaurant Locator Protocol	13	1091	27.7%

5.1.2 Partially Random Bits.

A more effective method of creating fuzz cases is to change only a few of the bits, instead of an average of half of them as in the fully random approach. There are two basic ways this can be achieved. Either a small subset of bits are randomly selected to flip, or a choice is made for each whether to flip or not with a low probability of a flip.

The latter approach was chosen because it proved far easier to implement. A random binary number is easier to generate in Microsoft Excel than a random draw from a set without replacement. Future work may involve mutations performed in this other way to see if it provides better coverage.

With the mutation randomization approach selected, two more questions were studied and answered to determine the optimal way to generate mutated messages. They were:

- When creating a series of mutated messages, which is more effective: mutating a valid message every time, or mutating the previously mutated message?
- What is the optimal probability of a bit flip?

5.1.2.1 Mutation Basis.

A case can be made for either answer to the first question. In support of mutating the previously mutated message, an issue with mutating a valid message every time is that the fuzzed message never strays too far from the valid one. Consider a valid message where the value in a four bit field is binary 0000. If the mutated messages are created by mutating this valid message each time, it is unlikely that all four bits will be mutated to test the value 1111. Later in this chapter the optimal bit flip probabilities are found for each of the three protocols; they range from 3% to 22%.

Therefore, in the most optimistic case, the odds this field will be mutated to 1111 are $0.22^4 = 0.0023$, or about twice in 1000 test cases.

If the previous mutated message is the basis for each new mutation, the message would morph over time into something different, and, it stands to reason, would cover test cases that would not be covered if mutation begins from the same message every time. This effect may provide better coverage.

The counter-argument to mutating the previously mutated message approach is that a chain of mutated messages will quickly become invalid. Like the disadvantage for completely random fuzzing discussed in Section 5.1.1, this approach will likely lead to messages with three or more invalid fields providing no coverage under ExCov. Once a few fields mutate to be invalid, every later message in the mutation change will also be invalid in those fields unless they are randomly made valid again. In practice this leads to long stretches of messages that provide no coverage because they have more than two invalid fields.

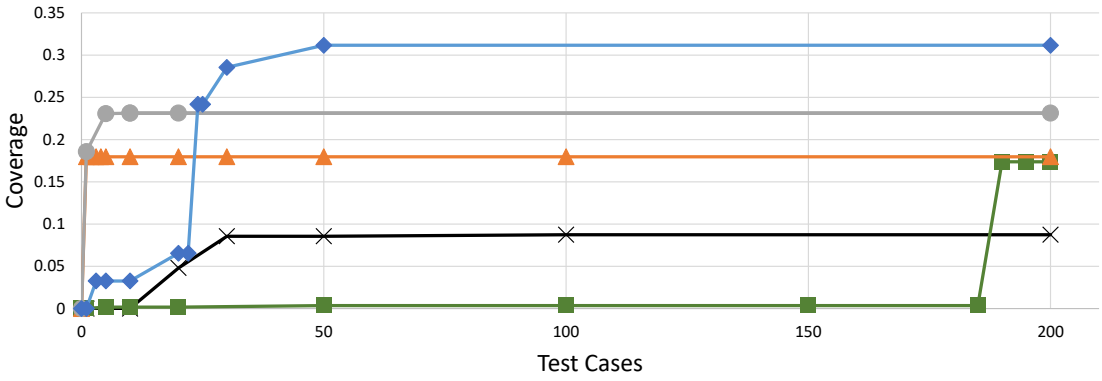

Figure 28. Coverage of test sets created using the mutation on mutation approach with five different original valid messages.

This effect is shown in Figure 28. To generate the Figure, the same starting valid message from the Restaurant Locator Protocol was mutated to get the first test case. Each subsequent test case was found by mutating the previous test case. Mutations were made on a bit by bit basis with a probability of a bit flip set such that the

expected number of bit flips per test case was one.

The figure shows five different mutation chains, with points representing the coverage of all test cases up to that point. It can be seen in the figure that at some point for each mutation chain, the coverage jumps. This is the point where the number of invalid fields is one, providing first and second order coverage. After the jump the message likely becomes too invalid to register any more coverage.

If an infinite number of test cases could somehow be tested, 100% coverage would be expected since every possible combination could be tested. However the largest tests run, 1000 test cases each, provided about the same amount of coverage, 20%, as test sets with 50 test cases. This pales in comparison to the coverage provided by random mutations from one valid test case. That figure was found to be 49.4%, the average of ten trials which ranged from 29% to 59%. This analysis conclusively shows that if mutating one valid test case, performing a random mutation on the valid test case each time provides more coverage than performing the mutation on the previous mutated case.

The tester will likely have more than one valid message to mutate however. And while it may be unproductive to mutate based on an already mutated message for hundreds of test cases, it may be useful to mutate the mutated message a few times, until it becomes too invalid to provide coverage. Consider this example. A tester takes 5 valid messages and mutates each of them by changing each bit with some probability. The tester then mutates these mutations in the same way, 9 times each, for a total of 10 mutated messages per valid message, or 50 total messages. For this case it can be said that the tester generated a test set using the mutation on mutation approach with 10 mutation rounds per valid message.

In this example, the approach taken may achieve better coverage than if the tester had taken one valid message and mutated it for 100 rounds. Not only will this

approach avoid long stretches of mutated messages that provide no coverage since each mutation set is capped at 10 rounds, but some diversity will also be added to the messages by having 10 different starting valid messages. One question that arises with this approach is "how many mutation rounds per valid messsage achieves optimal coverage?" If the answer is 1, then the mutation on mutation approach does no better than mutating a valid message each time.

To answer this question, 1000 valid messages were generated in the Restaurant Locator Protocol. Some of these messages were then selected, and mutated using the mutation on mutation approach for 1, 2, 4, 7, 10 and 15 rounds. Each number of rounds was tested with 10 test sets of 1000 test cases each. For example, to test the coverage of 10 rounds of mutation from random valid messages, 100 valid messages were selected, mutated 10 times each, and then the coverage for the test set was recorded. This was done for 10 such test sets. To test the coverage of 1 round of mutation (so really no mutation on mutation), all 1000 valid messages were mutated once for each of ten test sets.

The mean and standard deviation of the coverage distributions were then found for each number of mutation rounds. This result is shown in Figure 29. They show that even in this case, mutating the mutated message does not improve coverage. While the standard deviations around one to four mutation rounds suggests the observed downward trend in the means is not sure to hold at these values, certainly 10 and 15 mutation rounds provide far less coverage. There is no evidence here that any number of mutation rounds, other one round, is optimal. One mutation round is mutating from a valid message each time, so in this case mutation on mutation does not appear to be a useful approach.

The benefits of mutation on mutation fuzzing have not yet been entirely dismissed however. In mutative fuzzing the tester may not have access to randomly generated

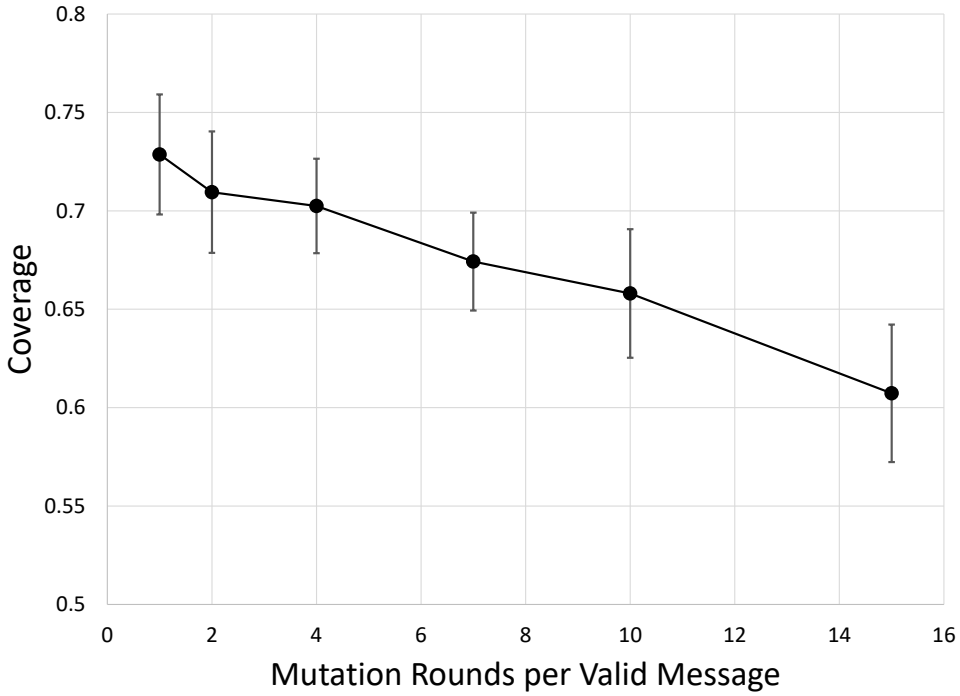

Figure 29. Coverage of test sets created by mutating from random valid messages 1 to 15 times. Each point is the mean of 10, 1000 test case trials. Bars denote one standard deviation from the mean for each set of trials.

valid messages like the ones used to generate Figure 29. The tester may have only a few, or even one valid message. If this is the case, it may yet be preferable to use the mutation on mutation approach rather than start from a small subset of valid messages each time.

To investigate this case, it was assumed that the tester has only one valid message. The coverage of tests sets where the mutations were applied to this valid message every time, and where the mutations were applied in a number rounds were compared. For example, test sets with four mutation rounds were created by taking a valid message, mutating it, mutating that mutation, etc. until four messages were generated. The process then restarted with the valid message, performed four mutations for four

more messages, and so on until 1000 messages had been created. The effect of bit flip probability on coverage was also of interest, so different values of that probability were tested against as well.

In all 170 test sets of 1000 test cases each were created, 10 for each of the 17 round — expected number of bit flips combinations. The average coverage of these test sets is shown in Figure 30. It turned out that if only one bit is expected to flip per message, 4 mutation rounds provides the best coverage. Overall however, one mutation round, or no mutation on mutation, between 5 and 7 expected bit flips per message provides the most coverage.

This leads to a final conclusion about the mutation on mutation approach. The only circumstance where it provides better coverage than mutating a valid message

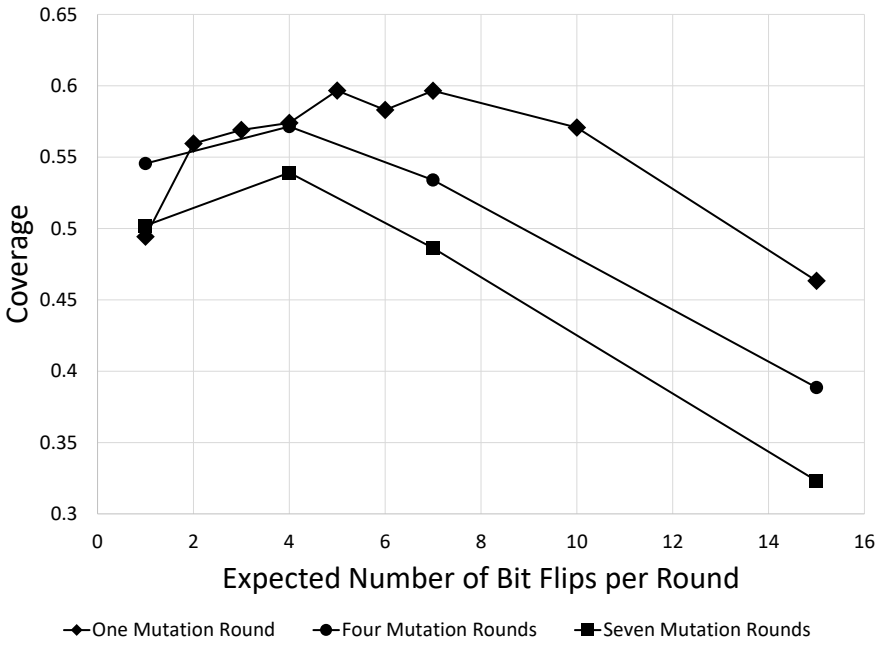

Figure 30. Average coverage versus expected number of bit flips for three mutation on mutation round lengths where all mutations start from same valid message. Each point is the average of 10 trials, each with 1000 test cases.

each time is when the probability of a bit flip is low. Since the tester generally has control over the probability of a bit flip in their mutator, the mutation on mutation approach is not more effective than the mutation on a valid message approach in any identifiable situation.

5.1.2.2 Optimal Bit Flip Probability.

The second question posed in this section is "what is the optimal probability of a bit flip?" Again there is a kind of balancing act between keeping a valid message valid by not mutating it enough, and making it too invalid by mutating it too much. If the expected number of bit flips in a mutation is low, then odds are better that the mutations that do occur will not render the valid message invalid. This means the mutated test case would provide no coverage. In contrast, if the expected number of bit flips in a message is high, the odds are better that more than two fields will become invalid and the test case will provide no first or second order invalid coverage. This means that there must be some optimal expected number of bit flips with respect to coverage.

It stands to reason that this optimal value is dependent on the protocol. If a protocol has a small amount of invalid space, it will require more bit flips to place a message in that invalid space. In a similar fashion, If a protocol has a large amount of invalid space, it should require less bit flips to create a message with one invalid field. Other properties of a protocol may also influence this optimal value, such as the number and size of the fields, and make-up of the valid messages being mutated from.

To find the optimal number of bit flips, test sets were generated by mutating randomly generated valid messages with varying expected number of bit flips. This was done for all three protocols.

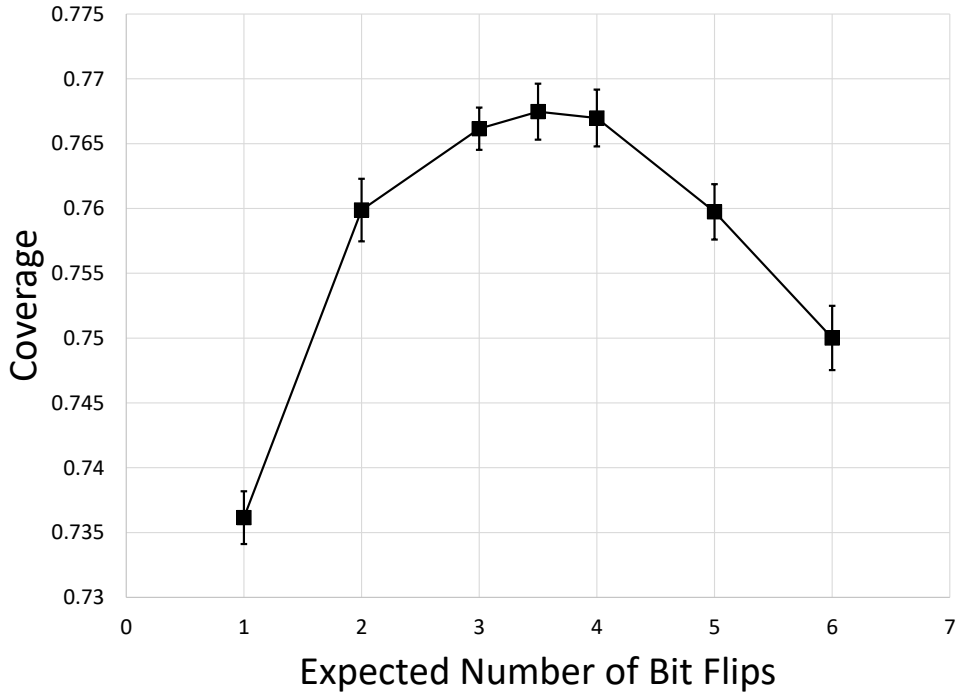

Figure 31. Average coverage for a mutation based fuzz test of the Simple Protocol based on bit flip probability. Each data point is the average of 10 trials of 1000 mutated test cases each. Bars denote one standard deviation from the mean for each set of trials.

- **Simple Protocol:**

  To test Simple Protocol, 70 test sets of 1000 test cases each were generated with various expected number of bit flip values. The results are shown in Figure 31. The points in this figure are the average coverage of 10 test sets with the listed expected number of bit flips. The bars denote standard deviation.

  Because it is so simple, Simple Protocol nicely illustrates the balancing act when it comes to bit flip probability. The figure shows a clear improvement in coverage as the expected number of bit flips approaches three, and a clear decline in coverage after the expected number of bit flips passes four. Small standard

deviations also validate these trends, and allow the assertion that an expected number of bit flips of 3.5 produces optimal coverage for mutative fuzzing of the simple protocol.

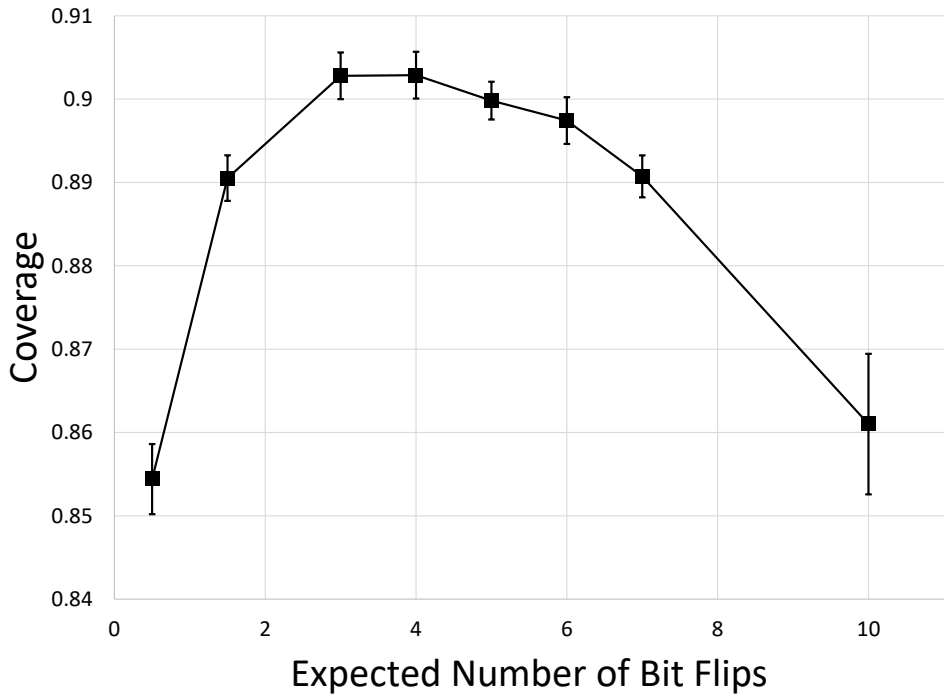

Figure 32. Average coverage for a mutation based fuzz test of the Meal Protocol based on bit flip probability. Each data point is the average of 10 trials of 1000 mutated test cases each. Bars denote one standard deviation from the mean for each set of trials.

- **Meal Protocol:**

  To test Meal Protocol, 80 test sets of 1000 test cases each were generated with various expected number of bit flip values. The results are shown in Figure 32. The points in this figure are the average coverage of 10 test sets with the listed expected number of bit flips. The bars denote standard deviation.

  The arc of this figure is similar to Simple Protocol, but has a less definitive peak.

Higher bit flip probabilities are also more effective than in Simple Protocol, perhaps a reflection on the relatively fewer invalid cases in Meal Protocol (see Table 5). The standard deviation is again fairly small compared to the jumps in the means across the figure. This again validates the observed curve, and suggests the peak wasn't created purely by chance. Oddly the best guess, based on this figure, for an optimum number of bit flips is again 3.5 flips.

- **Restaurant Locator Protocol:** To test Restaurant Locator Protocol, 90 test sets of 1000 test cases each were generated with various expected number of bit flip values. The results are shown in Figure 33. The points in this Figure

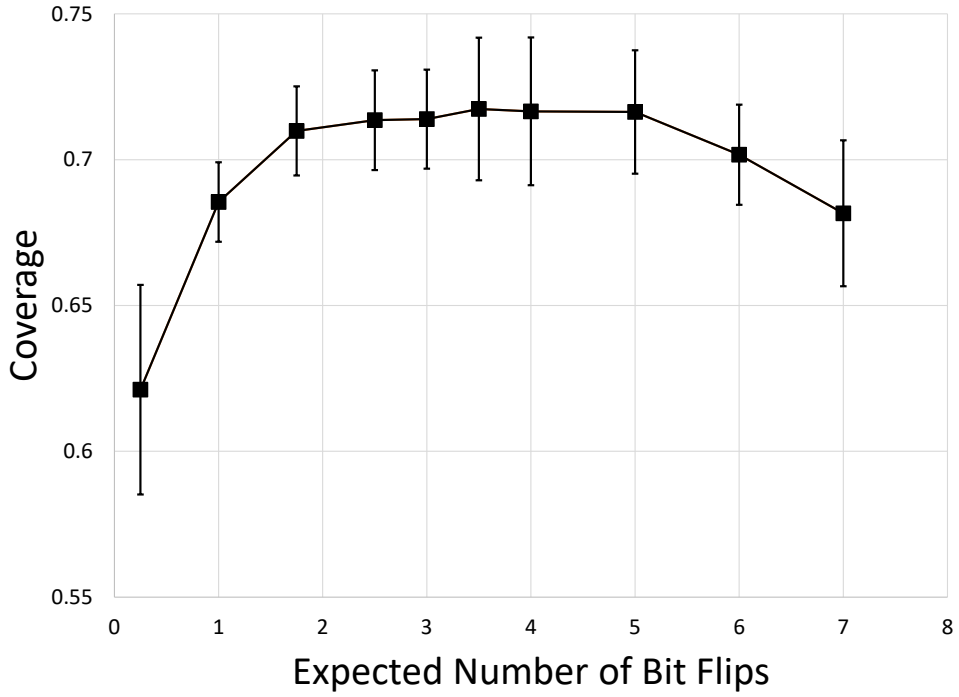

Figure 33. Average coverage for a mutation based fuzz test of the Restaurant Locator Protocol based on bit flip probability. Each data point is the average of 10 trials of 1000 mutated test cases each. Bars denote one standard deviation from the mean for each set of trials.

are the average coverage of 10 test sets with the listed expected number of bit flips. The bars denote standard deviation.

In contrast to the other two protocols, the RLP does not have a clear optimal number of bit flips. Not only are the averages all around 72% between 2 and 5 expected flips, but the standard deviations are large as well. This is surely a result of the increased complexity of this protocol. It features more fields and more invalid test cases than the other protocols, and also implements three choices and a rule set. Based on the fact that the maximum is also the mid point between 2 and 5, the apparent cut-offs for the peak of the curve, once again 3.5 bit flips is optimal.

Despite all three protocols having the same optimal expected number of bit flips, it is believed, based on the discussion presented earlier, that this value still depends on the protocol. The three invented protocols just happened to share the same value for the optimal number of expected number of bit flips.

5.2 Generative versus Mutative Fuzzing

Three methods for fuzz set generation have been presented thus far. Fully random mutative fuzzing was presented in Section 5.1.1, partially random mutative fuzzing with an expected number of bit flips per mutation was presented in Section 5.1.2, and generative fuzzing using GenFuzz was presented in Section 4.3. Each of these methods was used to generate test sets for the three protocols, and their coverage was found.

5.2.1 Simple Protocol.

Figure 34 shows the results of the three approaches to fuzz set generation for Simple Protocol. Like in previous figures, each point is the average of ten test sets

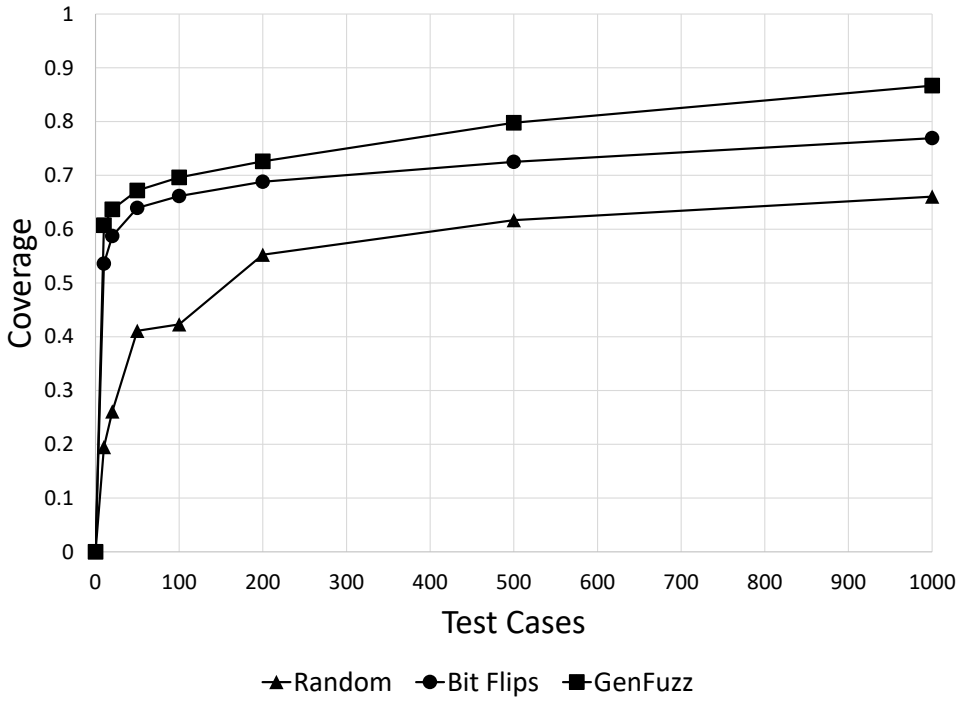

Figure 34. A comparison of three fuzz generation methods for the Simple Protocol

with the exception of GenFuzz, which has little variation between test sets due to its largely deterministic generation approach. Standard deviation is not shown for clarity, but is less than 10% for all points and less than 1% for points at 1000 test cases.

Not surprisingly, GenFuzz performs the best at any number of cases. The bit flip approach, using the optimal expected number of bit flips, 3.5, performs remarkably well also. This suggests that with the proper probability of bit mutation, simple protocols are well covered by mutative fuzzers.

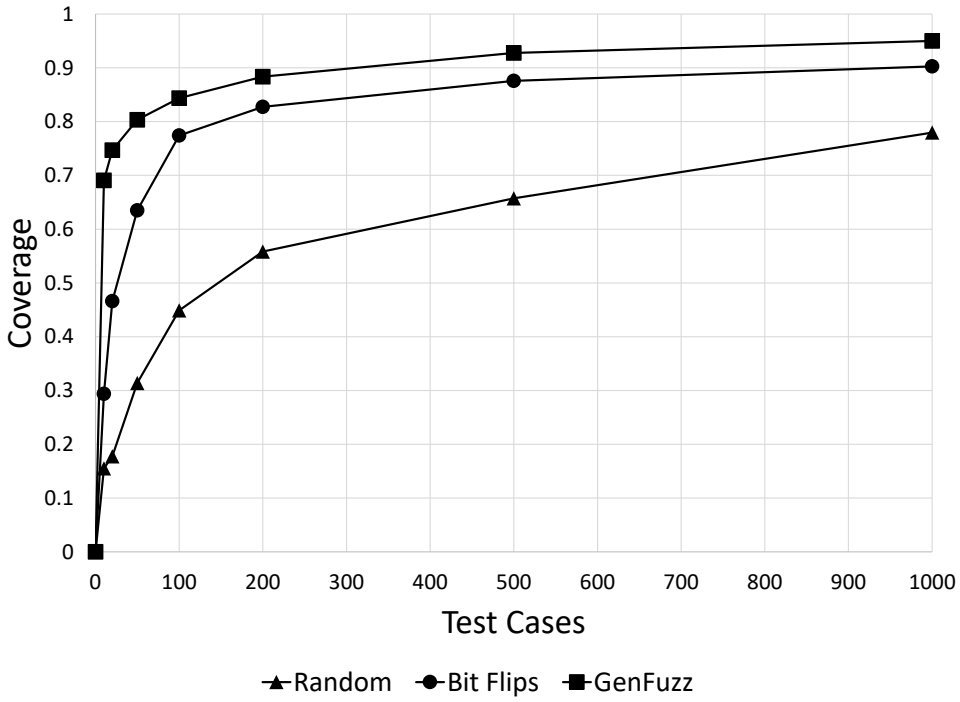

Figure 35. A comparison of three fuzz generation methods for the Meal Protocol

5.2.2 Meal Protocol.

Figure 35 shows the results of the three approaches to fuzz set generation for Meal Protocol. Like in previous figures, each point is the average of ten test sets with the exception of GenFuzz, which has little variation between test sets due to its largely deterministic generation approach. Standard deviation is not shown for clarity, but is less than 9% for all *Random* results, less than 14% for all *Bit Flip* results, and less than 1% for all *Bit Flip* results with 200 or more test cases.

Like with Simple Protocol, GenFuzz performs the best while an optimal number of bit flips does very well compared to random messages. Coverage for Meal Protocol is also very high. First order coverage actually reaches 100% from GenFuzz at 330

test cases. Its slow rise beyond that is caused by test cases that contribute second order coverage being added. Once again, an optimal number of bit flips provides almost as much coverage as GenFuzz. GenFuzz is by no means perfect, but based on its methodical switching between fields to fuzz, it would be very difficult to create a fuzzer that performs better for the first 100 test cases.

5.2.3 Restaurant Locator Protocol.

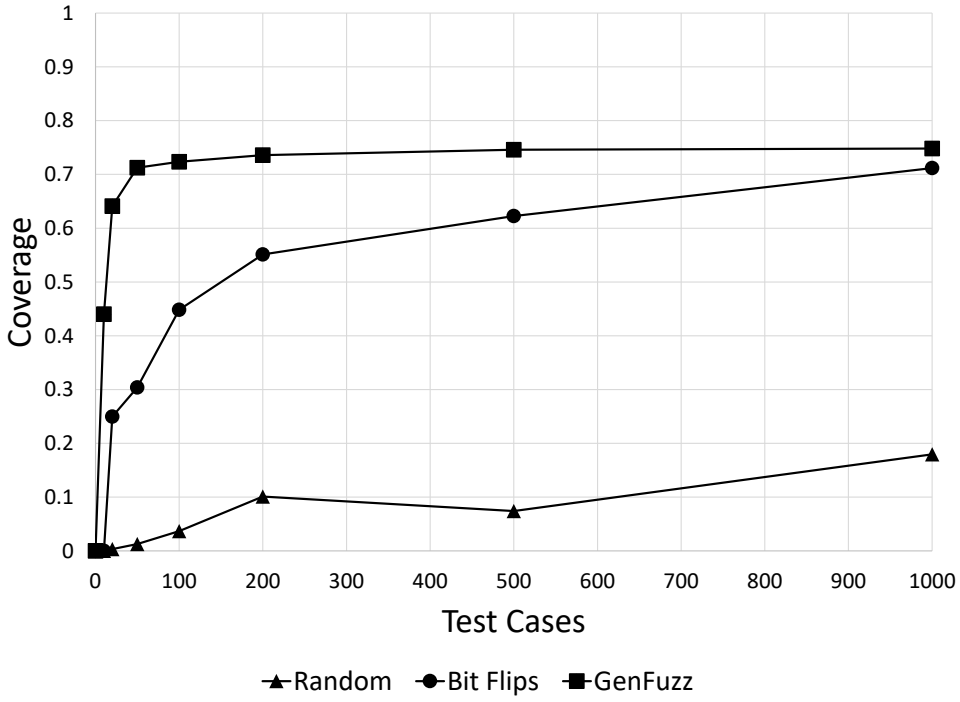

Figure 36. A comparison of three fuzz generation methods for the Restaurant Locator Protocol

Figure 36 shows the results of the three approaches to fuzz set generation for the Restaurant Locator Protocol. Like in previous figures, each point is the average of ten test sets with the exception of GenFuzz, which has little variation between test

sets due to its largely deterministic generation approach. Standard deviation is not shown for clarity, but is less than 10% for all *Random* results, less than 9% for all *Bit Flip* results, and less than 5% for all *Bit Flip* results with 50 or more test cases.

The Restaurant Locator Protocol is designed to be the most like actual military data link protocols, so its results may have the largest impact on actual fuzzing work. Due to the protocol's complexity, random fuzzing is barely effective. With standard deviations that are at times greater than the mean, the decrease in this curve at 500 test cases can be explained by random variation.

In contrast to the other protocols, the bit flip approach does not rival the GenFuzz approach until the number of test cases exceeds 500. This suggests that in cases where there is a limited number of test cases that can be tested against a system, a generative fuzzer will be far preferable to a mutative fuzzer. A Generative fuzzer that considers second order coverage (GenFuzz does not) may prove even more effective in this region.

At the opposite end of the figure, it appears that the Bit Flip method is set to surpass the GenFuzz method. This is expected to be the case if the methods had been tested further. GenFuzz is set up to focus on first order coverage. It achieves 100% first order coverage after 1091 test case for RLP. The fuzzer does not generate test cases that have more than one invalid field, so 100% second order coverage can never be achieved. Since the other two methods are purely probabilistic, any message is possible, therefore 100% coverage is possible with enough test cases. *Bit Flips* is bound to surpass the coverage of GenFuzz eventually. A fuzzer that considers second order invalid spaces will easily be able to out perform GenFuzz and the Bit Flip approach.

5.3 Generative versus Manual Fuzzing

Another way for a tester to generate test cases is to select them manually. Strictly speaking many may not consider this approach fuzzing since inputs are not automatically generated. In a way it more resembles a red-team vulnerability discovery approach. But it is possible for a tester to analyze a protocol and select cases thought to provide high coverage. It is also possible for the coverage calculator to compute a coverage value for these test cases.

A generative fuzzer has access to the data model, and should theoretically have access to all the information a tester has. For this reason, a generative fuzzer should always be able to be designed that provides coverage that meets or exceeds the coverage of a manually generated test set. But generative fuzzers aren't necessarily perfect, and GenFuzz certainly has some flaws that would allow a manually crafted test set to exceed its coverage.

The obvious drawback to manually crafting a test set is the time and effort required of the tester. GenFuzz generates thousands of test cases in seconds or less; it takes minutes to craft just a few messages manually even if the tester has some tools to help with the task.

To compare these two approaches, some fuzz test sets were crafted manually. Previously developed spreadsheets, created to test the various components of ExFuzz, were used allowing the messages to be generated with ease. The spreadsheets listed every field in a protocol and allowed entry of any desired value and include whatever field and words wanted in a message. They automatically generated the hexidecimal representation of each message, and some copy and pasting placed the test set into the text file format recognized by ExCov. This section compares these test sets with some generated by GenFuzz.

5.3.1 Simple Protocol.

Since the simple protocol is such a straightforward protocol it wasn't difficult to generate cases using a spreadsheet. In fact, generating all 690 first order invalid cases simply required using the Excel fill tool to create sequential lists of numbers. Each test case required one invalid field and two valid fields. The valid fields were selected using a random function, and each invalid value was tested only once.

Table 6. A comparison of manual test set generation and GenFuzz for Simple Protocol

Test Cases	Method	1st ord. cov.	2nd ord. cov.	Coverage	Generation Time
690	Manual	100%	68.1%	84.1%	7 min
	GenFuzz	100%	68.1%	84.1%	$<$ 1 sec
1000	Manual	100%	71.0%	85.5%	20 min
	GenFuzz	100%	73.3%	86.7%	$<$ 1 sec

The top half of Table 6 shows the coverage result for this manual test set. As designed, it achieved 100% first order coverage. GenFuzz also acheived this mark as it was designed to do. Second order coverage was also identical. This was no surprise since every test case contributed two invalid field combinations, the combinations that include the one invalid field. Since no invalid value was repeated, no field combination value was repeated either, guaranteeing equal coverage. The major difference between these two approaches is that GenFuzz took less than a second to generate these test cases while it took 7 minutes to craft them in Microsoft Excel.

Since these results provide little new information, another 310 test cases were generated to try to get as much second order coverage as possible. Since GenFuzz focuses on first order test cases, it does not create messages with two invalid fields. Therefore, in an attempt to outperform GenFuzz, a series of 310 messages with two invalid fields were created, roughly equally divided between the three field combinations.

When ExCov was run on this test set, it was surprising to discover that GenFuzz actually provided more coverage than the manual set did. After a little analysis it was

discovered why. When a message in Simple Protocol has one invalid field, say field A, it provides coverage for two field combinations, A-B and A-C. When a message has two invalid fields say A and B, it only provides coverage for field combination A-B. A-C and B-C are not covered because the other field not in the combination is invalid. This is dictated in the R^{th} order test case limitation in Section 3.2.2.2, on page 71.

This meant that every extra test case that was generated only provided coverage to one field combination. Every test case GenFuzz produced provided coverage for two field combinations unless the randomly selected pair of values happened to repeat. It is reasonable to think that testing some messages with two invalid fields should provide more coverage than testing more of the same valid-invalid combinations. The reason this does not provide more coverage is that a PMF specifically for vulnerabilities in second order spaces has not yet been developed. Such a PMF could cause the manually created test set to provide more coverage than GenFuzz because the manual approach created some field combinations with two invalid fields while GenFuzz did not.

5.3.2 Meal Protocol.

Meal Protocol was not simple enough to generate all first order values in a short amount of time as was the case with Simple Protocol. Instead each field with an invalid space was fuzzed, one at a time. Since the protocol involves three different words, it was simply set up so that the messages would rotate between the breakfast word, the lunch word and the dinner word. Care was taken to fuzz each field in a word once before repeating fields as GenFuzz did. Test set creation was stopped when each field with an invalid space small enough to manually cover had been covered, plus a few more cases with invalid values in the lager invalid spaces.

Table 7. A comparison of manual test set generation and GenFuzz for Meal Protocol

Test Cases	Method	1st ord. cov.	2nd ord. cov.	Coverage	Generation Time
24	Manual	80.9%	69.2%	75.0%	15 min
	GenFuzz	81.3%	70.3%	75.8%	$<$ 1 sec

The results are shown in Table 7. GenFuzz performed slightly better because it did not have a restriction on rotating between the three words. This allowed it to more evenly cover all the fields with invalid spaces, maximizing coverage. Again, the generation time is the major difference between the two fuzz set generation approaches.

5.3.3 Restaurant Locator Protocol.

Like the Meal Protocol, the Restaurant Locator Protocol is made up of a number of words. Unlike the Meal Protocol, more than one word may appear in a message. In fact, all 6 words can appear in the same message. To make the fuzz set generation easier, every message was set at the maximum valid length which contained all 6 words. The fuzzable fields were then made invalid one at a time for each test case. Once all fields with invalid spaces were fuzzed, each new message fuzzed a field for a second time. This is the same procedure applied in GenFuzz.

50 test cases were created, which took about ten minutes. It was decided to stop at this point because all the fields with smaller invalid spaces had been completely fuzzed, and only fields with very large invalid spaces remained. The results of this fuzzing are shown in Table 8.

Table 8. A comparison of manual test set generation and GenFuzz for the Restaurant Locator Protocol

Test Cases	Method	1st ord. cov.	2nd ord. cov.	Coverage	Generation Time
50	Manual	96.8%	67.8%	82.3%	10 min
	GenFuzz	97.3%	44.1%	70.7%	$<$ 1 sec
2000	GenFuzz	100%	53.2%	76.6%	5 sec

It was initially surprising to find that the manual test set had a higher coverage than GenFuzz's test set did. The number of generated test cases was increased to 2000 and GenFuzz was tested again to still find that its coverage was significantly below that of the 50 test case manual set. It was noticed that GenFuzz actually outperformed the manual generation method in first order coverage, but trailed far behind in second order coverage.

It was determined after some analysis that the reason for this was the choice to use the maximal length message every time. GenFuzz is set up to randomly select message length along with the other valid values it selects for each test case. This means that most messages in the GenFuzz test set are shorter than in the manual test set.

A longer message includes more field combinations, so every test case in the manual set guaranteed a near maximal amount of second order coverage. Since no two invalid values were repeated, no field combination values were repeated either. Also since each field with an invalid space was fuzzed, almost every possible field combination was fuzzed at least once. This provided a major boost to second order coverage.

GenFuzz relies on randomly including the necessary words to test each field combination. Even with 2000 test cases many field combinations went untested. This allowed the manual generation method with 50 test cases to provide more coverage than GenFuzz at 2000 test cases. The biggest take away here is that a generative fuzzer ought to take advantage of long message lengths when it can to increase second order fuzzing.

VI. Future Work and Conclusion

6.1 Future Work

Throughout this thesis references have been made to problems that could have been better solved with more time, or issues that could and ought to be address with future work. A compilation of these potential endeavors that would contribute more to this work is presented here.

- **Improve Vulnerability PMFs:** A theme touched on at many points in this thesis is the ExCov criterion's reliance on a probabilistic model of the nature of vulnerabilities in general. The model used in this thesis, created in Section 3.3 using data from Appendix D, is developed from real world protocols, but not real world vulnerabilities. Surveying real vulnerabilities across data link like protocols and incorporating the information gained into the vulnerability models and then the PMFs will greatly improve the legitimacy of the coverage criterion as a whole.

- **Develop a field combination coverage equation:** In Chapter III, the equations that take a set of invalid test cases and return a coverage value are all based on first order fields. When the calculation moves to multi-order field combinations, the old coverage equation is used for convenience. This may not be wrong exactly, with a unique PMF an argument could be made that this is the proper equation, but the uniqueness of field combinations compared to fields suggest that another derivation unique to multi-order spaces should be derived. This may be a complex effort, deriving the equation for a single field was no small feat, and multi-order field combinations are far more convoluted than single fields.

- **Add capabilities to ExCov:** In Section 4.2 on page 108 some capabilities missing from ExCov are listed. Some of these exist in Chapter III and just need to be added to the tool, others need theory to be developed and added to the coverage criterion. Doing this for these features will improve ExCov's applicability and usability.

- **Improve GenFuzz:** In Chapter V, there are a number of mentions of the potential to improve the generative fuzzer application GenFuzz. GenFuzz's main failing is that it focuses only on first order and not second order test cases. Another generative fuzzer could be designed that focuses on both orders to maximize coverage. Aspects of GenFuzz could also be improved, like adding support for weighted fields or values, or unique fuzzing approaches for numeric and basic fields.

- **Investigate other mutation approaches:** Mentioned in Section 5.1.2 is the potential to study alternative methods of mutative fuzz set generation. The method used in this thesis is to mutate each bit with some probability set such that the expected total number of mutated bits in a message is a set value. This means that a message may experience no mutations, or many more mutations than intended. An alternative approach to mutation would set the number of mutations per message at a fixed value, randomly selecting the appropriate number of bits for mutation. Testing whether or not such an approach provides more or less coverage could help designers of mutative fuzzing tools make better decisions about their fuzzer's operation.

- **Test ExFuzz on open source protocols:** In the unlimited distribution portion of this thesis, ExFuzz was only ever applied to protocols crafted specifically for this thesis. Testing the tool against real protocols should highlight

its strengths and weaknesses and suggest further areas of improvement. It may also uncover some vulnerabilities in these protocols.

- **Improve the usability of ExFuzz:** Since ExFuzz is currently a Microsoft Visual Studio C++ project, it is not very user friendly or distributable. For this tool to be ultimately useful, it needs to feature a user friendly interface and clear documentation. These could be developed for ExFuzz, or ExFuzz could be integrated within some other fuzzing tool. From its conception, ExFuzz was designed to have some similarities to Peach, namely a Peach Pit, to support integration with that tool in the future.

6.2 Conclusion

Negative testing, specifically fuzz testing, is crucial to defending weapon systems from cyber attacks. To implement fuzz testing effectively across all weapon system components, a way to measure the effectiveness of any test is needed. This thesis proposed such a metric, called a coverage criterion.

Chapter II looked at the state of the art in fuzz testing and a few attempts to apply some sort of metric to a fuzz test. The chapter concluded that none of these efforts produced a metric that could be easily understood by a tester without intricate knowledge of the criterion's construction. In this chapter some open source fuzzing frameworks, Peach and Sulley, were investigated to see how their strengths might be leveraged to create a tool that can measure coverage.

Chapter III developed equations and procedures that allow computation of the coverage of a set of fuzz test cases using this thesis's proposed definition for coverage: *the expected percent of existing vulnerabilities discovered by a set of test cases.* The method was named ExCov, short for expected coverage, because it relies heavily on probabilistic expectation, and also because it means what a tester would expect it to

mean.

With the criterion developed, a suite of tools built around the method was presented in Chapter IV. This tool suite was called ExFuzz. ExFuzz consists of three main parts: a data model parser called DataModel that interperates a Peach Pit data model with some custom modifications; a coverage calculator called ExCov that implements the new criterion; and a generative fuzzer called GenFuzz that produces a high coverage test set for any data model based on the coverage criterion.

Finally, Chapter V used ExFuzz to compare a variety of fuzz test set generation approaches. By doing this, some questions were answered regarding which generations methods are the most effective and by how much. These results can be used to develop and execute better fuzz tests. By exercising the ExFuzz tool, the implementability of the ExCov criterion was verified. This chapter also demonstrated how the tool can be used to create high coverage test sets with GenFuzz.

Fuzz testing is in many ways a very imprecise science. The possible number of inputs nearly always borders on infinite and despite the amount of testing done, there is almost always the possibility that a vulnerability has been missed. Without an effective way to measure the quality of a set of fuzz test inputs, this technique will remain imprecise, leaving testers with a great amount of uncertainty despite the number or quality of fuzz tests they apply. By introducing such a method, fuzz testing techniques may be improved so that they may be used with confidence to defend or most critical systems.

Appendix A. Terms and Definitions

This appendix contains two tables. The first lists some acronyms used throughout the thesis,and their meaning. The second table lists a variety of terms used throughout the thesis that have a non-obvious meaning, or may require a precise definition. These terms are accompanied by a description of the term, its importance and a precise definition if necessary.

Table 9. A list of the acronyms used throughout this thesis and their definitions

Acronym	Description	First Mention
GCD	Greatest Common Divisor	Sec. 4.3, Pg. 110
GPS-SPS	Global Positioning System - Standard Positioning Service	Ch. I, Pg. 18
IIC	Invalid in Combination	Sec. 3.2.2, Pg. 70
IPv4	Internet Protocol Version 4	Sec. 4.1, Pg. 98
MIL-STD	Military Standard	Sec. 4.1, Pg. 98
PMF	Probability Mass Function	Sec. 3.1.2, Pg. 45
RLP	Restaurant Locator Protocol	Sec. B.3, Pg. 149
SUT	System Under Test	Ch. I, Pg. 14
SVCov	Semi-valid Input Coverage	Sec. 2.2.2, Pg. 29

Table 10. A list of key terms used throughout this thesis and their definitions

Term	Description	Reference
Adequacy Criterion	The thing or things that are measured to determine the adequacy of a test	Sec. 2.2.1,Pg. 28
Agent	An entity responsible for monitoring the SUT for any unexpected behavior during a fuzz test.	Sec. 1.4, Pg. 17
Choice	An XML element used in a Peach Pit that allows the value in one field determine which set of fields appears elsewhere in the message.	Sec. 2, Pg. 101

Term	Description	Reference
Choice Relation*	A type of field in a Peach Pit data model that dictates the result of a choice elsewhere in the model using a relation child element.	Sec. 4.1.2, Pg. 104
Coverage	A term generally understood to mean the extent to which a test set covers the space of inputs in a fuzz test. This thesis proposes a specific definition for this term as: *the expected percent of existing vulnerabilities discovered by a set of test cases.*	Sec. 1.5, Pg. 20
Coverage Calculator*	An application that computes the effectiveness of a fuzz test set based on a coverage criterion.	Sec. 1.4, Pg. 17
Coverage Criterion	An adequacy criterion that uses a measurement of coverage to determine the adequacy of a test.	Sec. 1.3, Pg. 14
DataModel (C++ tool)	The C++ tool developed as part of ExFuzz that builds a data model object from a modified Peach Pit XML document	Sec. 4.1.3, Pg. 105
DataModel (XML)	A Peach Pit XML file that is modified with the extensions shown in Section 4.1.2.	Sec. 4.1, Pg. 98
Discover*	A test case is said to discover some vulnerability V if it is the first test case in a test set to trigger V.	Sec. 3.1, Pg. 34
Disjoint*	A numeric field is disjoint if the values of its invalid space is not contiguous.	Sec. 3.1.3.2, Pg. 52
ExFuzz*	Expected Fuzzer. This is the C++ project that combines the three tools created to demonstrate the ExCov coverage criterion.	Ch. IV, Pg. 97
ExCov*	Expected Coverage. This term can refer to either the coverage criterion presented in this thesis, or the coverage calculator that implements the criterion.	Ch. III, Pg. 33; Sec. 4.2, Pg. 106
Field	The fundamental element of a protocol. Every bit in a message belongs to a field, and every field has a length (in bits) and a valid space.	Sec. 3.2.1, Pg. 63

Term	Description	Reference
Field Combination*	A set of two or more fields. Vulnerabilities reside in field combinations instead of fields when they depend on the values of multiple fields.	Ch. III, Pg. 33
Field Coverage*	A measure of coverage for a single field in a protocol. This is an intermediate result in the computation of the coverage of a test set. Field Coverage is explicitly defined as *The expected percentage of existing vulnerabilities in a field discovered by a set of test cases.*	Sec. 3.1, Pg. 34
Fuzzing	A method of vulnerability discovery where little is known about the inner workings of the system under test. According to [22], "Fuzzing is the the process of sending intentionally invalid data to a product in the hopes of triggering an error condition of fault."	Sec. 1.3, Pg. 14
Generative Fuzzing	A method of test set generation where invalid messages are generated based on a protocol model or data model.	Sec. 1.4, Pg. 16
GenFuzz*	Generative Fuzzer. The C++ tool developed as part of ExFuzz that generates a set of test cases from a data model that acheives high coverage as measured by the ExCov criterion.	Sec. 4.3, Pg. 109
Invalid in Combination (IIC)*	A set of values for a field combination that are valid independently, but invalid when combined.	Sec. 3.2.2.1, Pg. 70
Invalid Space*	The set of values that do not meet the requirements of the protocol specification for a field or field combination	Sec. 3.1.1, Pg. 36
Mutative Fuzzing	A method of test set generation that involves taking a valid message and changing some of the bits to make it invalid. Mutative fuzzing does not require a protocol model or, in some cases, any knowledge of the protocol whatsoever.	Sec. 1.4, Pg. 16

Term	Description	Reference
Numeric Field*	A type of field where invalid values have an implied numeric meaning. An important characteristic of this field type is that vulnerabilities are assumed to always be contiguous in the invalid space.	Sec. 3.1.3.2, Pg. 50
Order*	The number of fields being assessed or fuzzed. An N^{th} order vulnerability spans N fields, and N^{th} order coverage refers to the coverage of N^{th} order vulnerabilities.	Ch. III, Pg. 34
Protocol Model	A general term for what is referred to as a data model in this thesis. See data model (XML)	Sec. 1.4, Pg. 16
Repeating Relation	A type of field in the Peach Pit data model that sets the number of times a block of fields repeats.	Sec. 3, Pg. 101
Specific Vulnerability Space*	A set of values that trigger one specific vulnerability. A specific vulnerability space is always a subset of a field's vulnerability space.	Sec. 3.1.1, Pg. 37
Structure Field*	A field that contains information about how a message is structured or interpreted. There are three types of structure fields used in this thesis, the repeating relation field, the choice relation field, and the token field.	Sec. 3.1.3.1, Pg. 48
System Under Test (SUT)	The system that is being tested for vulnerabilities. In fuzz testing, this system receives the fuzzed messages and is monitored for faults.	Sec. 1.4, Pg. 17
Test Case	One message, or instance of a protocol, applied to the SUT during a fuzz test.	Sec. 3.2.1, Pg. 63
Test Set	A set of messages, or test cases, to be applied to a SUT in a fuzz test. The coverage calculator computes to coverage of this set of messages.	Sec. 1.4, Pg. 16
Tester	The human designing, running, and observing the test.	Sec. 1.3, Pg. 14
Token	A type of field in a Peach Pit data model that dictates the selection made by a choice block. Token fields are always within their choice block in contrast to choice relation fields.	Sec. 3.2.3.2, Pg. 76

Term	Description	Reference
Valid Space*	The set of values that meet the requirements of the protocol specification for a field or field combination	Sec. 3.1.1, Pg. 36
Vulnerability	According to Microsoft, a vulnerability is "a weakness in a product that could allow an attacker to compromise the integrity, availability, or confidentiality of that product." This definitions suits this thesis's needs fine, with the product being the SUT.	[20]
Vulnerability Space*	A set of values that is the union of all specific vulnerability spaces of a field. A vulnerability space is always a subset of an invalid space.	Sec. 3.1.1, Pg. 37
* Denotes terms created in the context of this thesis.		

Appendix B. Protocol Examples

This appendix describes the fictitious protocols that were written to faciliate development of a coverage criterion, and provide insight into different methods of fuzzing. The protocols were designed with varying levels of complexity. The simplest protocol, termed *Simple Protocol*, was used to confirm basic coverage calculations by hand. Two, more complex, protocols included some features that were more difficult for the ExFuzz suite to handle, and were designed to closely resemble military data link protocols. In order of complexity, these are *Meal Protocol* and *Restaurant Locator Protocol*.

In this appendix, each protocol will be described as it might appear in a protocol standard document. Then the Data Model, presented in section 4.1, that describes the protocol will be presented. These data models have been supplied as inputs to the ExFuzz tools to produce the results in Chapter V.

B.1 Simple Protocol

The simple protocol consists of three, eight bit fields. All three fields have the same rules and meaning. They must contain an ASCII uppercase letter, that is a value between decimal 65 (A) and 90 (Z). This means that the decimal values 0 to 64 and 91 to 255 are invalid. This protocol could be imagined as part of a larger system, perhaps passing three letter error or status codes, or maybe domestic airport codes. Figure 37 shows the bit map for the Simple Protocol.

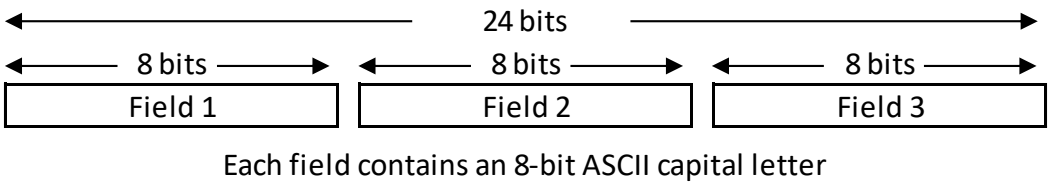

Figure 37. Bit map for the Simple Protocol

This protocol is represented by the modified Peach Pit Data Model shown here:

```
<DataModel name="SimpleProtocol">
        <Number name="LTR1" size="8">
                <Hint valid ="65-90"/>
        </Number>
        <Number name="LTR2" size="8">
                <Hint valid ="65-90"/>
        </Number>
        <Number name="LTR3" size="8">
                <Hint valid ="65-90"/>
        </Number>
</DataModel>
```

B.2 Meal Protocol

The meal protocol is imagined as a protocol that transfers a meal order, perhaps from a customer to the kitchen. The protocol is organized into bytes, and a message is 2 to 8 bytes long. The message structure is shown in Figure 38.

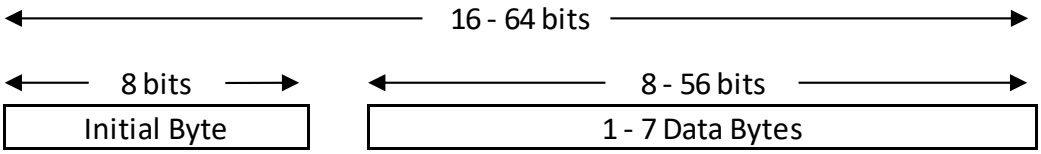

Figure 38. The structure of a message in the Meal Protocol

The first byte in every message is called the initial byte and carries meta data about the order such as the order number, the meal, (breakfast, lunch, or dinner) and the number of people served by the order. The remaining bytes convey the drink, meal, and dessert orders. Since the food is different depending on which meal is being

served, the definition of these data bytes depends on the meal in the initial byte field. The bit maps for the initial and data bytes are shown in Figure 39. Table 11 describes each field and lists their valid values.

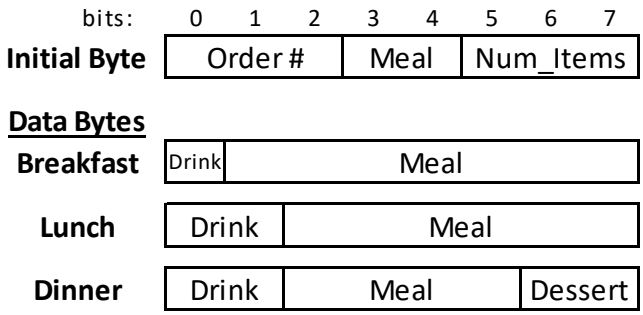

Figure 39. Bit maps for the bytes in the Meal Protocol

Field Name	Description	Length (bits)	Valid Values (decimal)
Order #	An identification number for the order	3	0 - 7
Meal (Initial Byte)	Selects the type of meal, Breakfast (0), Lunch (1), or Dinner (2)	2	0 - 2
Number of Items (Num_Items)	Indicates the number of data bytes in the message. This can be thought of as the number of customers on the order.	3	1 - 7
Drink (Breakfast)	Water (0) or Coffee (1)	1	0 or 1
Meal (Breakfast)	There are 15 breakfast items on the menu. The value here indicates which item is requested.	7	1 - 15
Drink (Lunch)	Water (0), Soda Pop (1), or Juice (2)	2	0 - 2
Meal (Lunch)	There are 21 lunch items on the menu. The value here indicates which item is requested.	6	1 - 21
Drink (Dinner)	Water (0), Soda Pop (1), or Juice (2)	2	0 - 2

Field Name	Description	Length (bits)	Valid Values (decimal)
Meal (Dinner)	There are 16 dinner items on the menu. The value here indicates which item is requested, with item 16 being represented as decimal 0.	4	0 - 15
Dessert	There are 4 desserts on the menu. The value here indicates which dessert is requested, with dessert 4 being represented as decimal 0.	2	0 - 3

Table 11. A description of the fields in the Meal Protocol

This protocol is represented by the modified Peach Pit Data Model shown here:

```
<DataModel name="MealOrder">
    <Number name="OrderNumber" size ="3" value="0">
        <Hint valid ="all"/>
    </Number>
    <Choice name="MealTypeChoice">
        <Block name="Breakfast">
            <Number name = "BreakfastType" token="true" size="2" value="0">
                <Hint priority="5"/>
            </Number>
            <Number name="BNum_Items" size="3">
                <Relation type="count" of="BreakfastItem" adjustment="0"/>
                <Hint valid ="1-7"/>
            </Number>
            <Block name="BreakfastItem" minOccurs="0" maxOccurs="7">
                <Number name="BDrink" size="1">
                    <Hint valid ="all"/>
                </Number>
                <Number name="BMeal" size="7">
                    <Hint valid ="1-15"/>
                </Number>
            </Block>
        </Block>
        <Block name="Lunch">
            <Number name = "LunchType" token="true" size="2" value="1"/>
            <Number name="LNum_Items" size="3">
                <Relation type="count" of="LunchItem" adjustment="0"/>
                <Hint invalid ="0"/>
            </Number>
            <Block name="LunchItem" minOccurs="0" maxOccurs="7">
                <Number name="LDrink" size="2">
                    <Hint invalid ="3"/>
                </Number>
                <Number name="LMeal" size="6">
                    <Hint valid ="1-21"/>
                </Number>
            </Block>
        </Block>
        <Block name="Dinner">
            <Number name = "DinnerType" token="true" size="2" value="2"/>
            <Number name="DNum_Items" size="3">
                <Relation type="count" of="DinnerItem" adjustment="0"/>
                <Hint invalid ="0"/>
            </Number>
            <Block name="DinnerItem" minOccurs="0" maxOccurs="7">
                <Number name="DDrink" size="2">
                    <Hint invalid ="3"/>
```

```
                </Number>
                <Number name="DMeal" size="4">
                    <Hint valid ="all"/>
                </Number>
                <Number name="DDessert" size="2">
                    <Hint valid ="all"/>
                </Number>
            </Block>
        </Block>
    </Choice>
</DataModel>
```

B.3 Restaurant Locator Protocol

Restaurant Locator Protocol (RLP) is imagined as a protocol that delivers information about restaurants anywhere in the world. A RLP message consists of one to six words. The first word must always be the same word, called Word 1. Following Word 1, Words 2, 3, and 4 may appear in any order with the following caveats:

- Word 2 must always be immediately followed by Word 3, and Word 3 must always be preceded by Word 2. This is because these words convey latitude and longitude, and it would not make sense for one to exist without the other.
- Words 2 and 3 may not appear more than once in a message. A restaurant can not be located at two different points.
- Word 4 may not be repeated more than 3 times.

Each word is 16 bits long. Since a message may be from one to six words in length, it may be from 16 to 96 bits in length. Figure 40 shows the structure of an RLP message.

The first three bits of each word define which word it is. The remaining 13 bits convey some information about the restaurant associated with the message. Word 4 also can be one of three different types, depending on the value bits 3 - 5. The remaining fields in the word depend on this value. Word 4 type 3 has another quirk where some fields depend on the value of bit 6. If bit 6 is a 0, bits 7 - 14 are one field

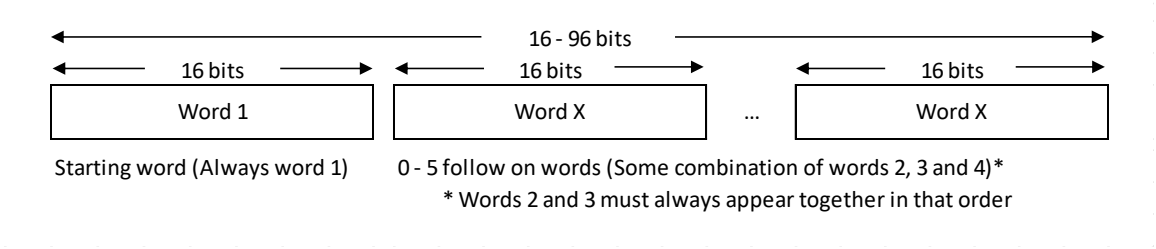

Figure 40. The structure of a message in the Restaurant Locator Protocol

representing a name. If bit 6 is a 1, bit 7 indicates a gender (either male or female) and bits 8 - 14 are a field representing a name. Figure 41 shows a bit map of the fields in all the words of RLP. Table 12 describes each field and lists their valid values.

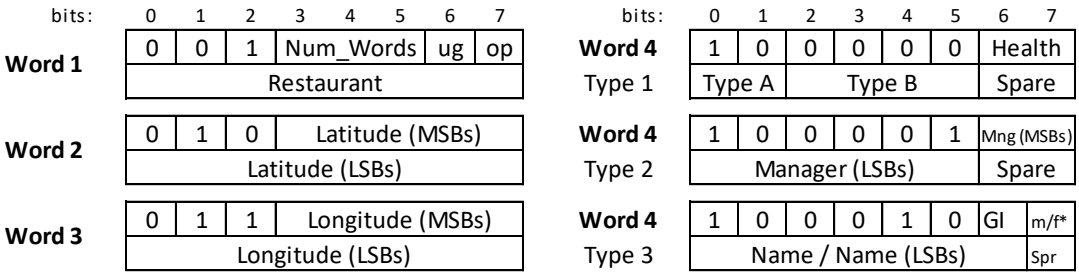

Figure 41. Bit maps for the words in the Restaurant Locator Protocol

Field Name	Description	Length (bits)	Valid Values (decimal)
Number of words (Num_Words)	Describes the number of words following Word 1 in the message	3	0 - 5
Urgent Flag (ug)	Indicates if the message is urgent	1	0 or 1
Open Flag (op)	Indicates if the restaurant is open	1	0 or 1
Restaurant	Indicates the restaurant name. This would involve a table (not included) that might start, 0 – Unknown, 1 – Einstein Bagels, etc.	8	0 - 162
Latitude	Indicates the latitude in 2's compliment and increments of $^{90}/_{4095}$ degrees. The polar regions (above 80°)are considered invalid and shall not be used	13	455-7736
Longitude	Indicates the longitude in 2's compliment and increments of $^{180}/_{4095}$ degrees.	13	0-8191
Healthiness (Health)	Indicates the healthiness of the food on a scale from zero to three	2	0-3
Type A	0 – Fast food 1 – Casual sit down 2 – Upscale	2	0-2

Field Name	Description	Length (bits)	Valid Values (decimal)
Type B	0 – No Type 1 – American 2 – Itallian 3 – Mexican 4 – Chinese 5 – Japanese/Korean 6 – Mediterranean 7 – Middle Eastern 8 – Indian 9 – French 10 – German 11 – Pizza 12 – Sandwiches 13 – Buffet	4	0-13
Manager	A name for the manager of the restaurant. As with the restaurant field, this may involve a table.	8	0-254
Gender Indicator	Determines which fields appear next. If a 0, the 8 bit name field appears, if a 1, the 1 bit gender field, and the 7 bit gendered name field appear.	1	0 and 1
Name	A name of a restaurant employee, as with the manager field, this may involve a table.	8	0-254
Male or Female (m/f)	A male (0) or female (1) name follows	1	0 and 1
Gendered Name	A name of a restaurant employee, as with the manager field, this may involve a table.	7	0-126
Spare (spr)	Unused bits	varies	0

Table 12. A description of the fields in the Restaurant Locator Protocol

This Restaurant Locator Protocol is represented by the modified Peach Pit Data Model shown here:

```
<DataModel name="Restaurant_Locator">
    <Block name="OuterRepeatingBlock" minOccurs="1" maxOccurs="8">
        <RuleSet block="OuterRepeatingBlock" choice="Word_Choice">
            <Position token="0" present="yes" position="1" />
            <Repeats token="0" precision="exactly" times="1" />
            <Repeats token="1" precision="no more than" times="1" />
            <Repeats token="2" precision="no more than" times="1" />
            <Repeats token="3" precision="no more than" times="3" />
            <Sequence token="1" token2="2" present ="yes" bora="after" proximity="immediately" />
            <Sequence token="2" token2="1" present ="yes" bora="before" proximity="immediately" />
        </RuleSet>
        <Choice name="Word_Choice">
            <Block name="Word1">
                <Number name = "Word_Type1" token="true" size="3" value="1">
                    <Hint weight="10"/>
                </Number>
                <Number name="Num_Words" size="3">
                    <Hint valid ="0-5"/>
                    <Relation type="count" of="OuterRepeatingBlock" adjustment="+1"/>
                </Number>
                <Number name="Urgent" size="1">
                    <Hint valid ="all"/>
                </Number>
                <Number name="Open" size="1">
                    <Hint valid ="all"/>
                </Number>
                <Number name="Restaurant" size="8">
                    <Hint valid ="0-162"/>
                </Number>
            </Block>
            <Block name="Word2">
                <Number name = "Word_Type2" token="true" size="3" value="2">
                    <Hint weight="10"/>
                </Number>
                <Number name="Latitude" size="13">
                    <Hint valid ="455-7736" type="numeric"/>
                </Number>
            </Block>
            <Block name="Word3">
                <Number name = "Word_Type3" token="true" size="3" value="3">
                    <Hint priority="5"/>
                </Number>
                <Number name="Latitude" size="13">
                    <Hint valid ="all"/>
                </Number>
            </Block>
            <Block name="Word4">
                <Number name = "Word_Type4" token="true" size="3" value="4">
                    <Hint weight="10"/>
                </Number>
                <Choice name="label_choice" >
                    <Block name ="Word4a">
                        <Number name="Labela" size="3" token ="true" value ="0">
                        </Number>
                        <Number name="Healthiness" size="2">
                            <Hint valid ="all"/>
                        </Number>
                        <Number name="TypeA" size="2">
                            <Hint invalid ="3"/>
                        </Number>
                        <Number name="TypeB" size="4">
                            <Hint valid ="0-13"/>
                        </Number>
                        <Number name="Spare" size="2">
                            <Hint valid ="0"/>
```

```
                    </Number>
                </Block>
                <Block name ="Word4b">
                    <Number name="Labelb" size="3" token ="true" value ="1">
                    </Number>
                    <Number name="Manager" size="8">
                        <Hint valid ="0-254"/>
                    </Number>
                    <Number name="Spare" size="2">
                        <Hint valid ="0"/>
                    </Number>
                </Block>
                <Block name ="Word4c">
                    <Number name="Labelc" size="3" token ="true" value ="2">
                    </Number>
                    <Number name = "Gender_inc" size="1">
                        <Relation type="choice" of="Word4cChoice" links="0,1"/>
                    </Number>
                    <Choice name="Word4cChoice">
                        <Block name="NoGender">
                            <Number name="StaffName1" size="8">
                                <Hint valid ="0-254"/>
                            </Number>
                        </Block>
                        <Block name="Gender">
                            <Number name="Manager" size="1">
                                <Hint valid ="all"/>
                            </Number>
                            <Number name="StaffName2" size="7">
                                <Hint valid ="0-126"/>
                            </Number>
                        </Block>
                    </Choice>
                    <Number name="Spare" size="1">
                        <Hint valid ="0"/>
                    </Number>
                </Block>
            </Choice>
        </Block>
    </Choice>
</Block>
</DataModel>
```

Appendix C. Coverage Calculator Application Example

In this appendix, the equations and procedures presented in Chapter III will be applied to a simple protocol and test set in order to demonstrate how they are properly applied.

The protocol used in this chapter was created to illustrate all the major elements a data model may contain, while still being short enough to present in this form. The protocol consist of six fields, named fields A through F, and can be represented by the diagram in Figure 42.

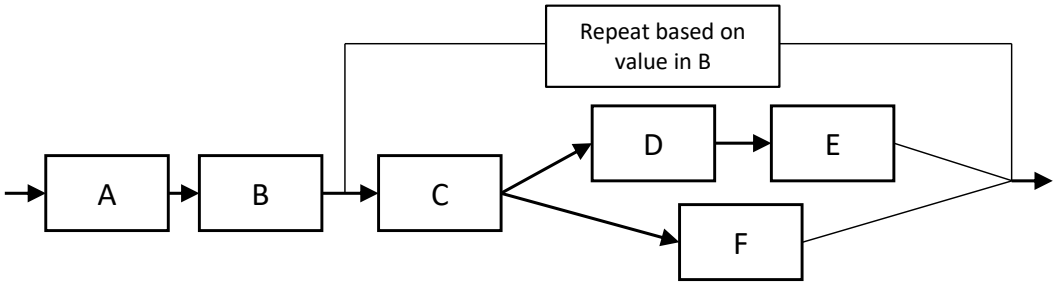

Figure 42. Diagram of the fields in the example protocol

Field B is a repeating relation field that says how many times the block of fields C, D, E, and F can repeat. Field C is a choice relation field that determines whether fields D and E, or F are included. Details of the fields are shown in Table 13. Since there is a choice within a repeating block, a set of rules may be applied that restrict what choice can be made when in a sequence of repeating choices. For this protocol the only rule is that a choice of 0, or fields D and E must follow field C, must always be the first choice in a sequence of repeated choices. After that either choice may be made.

Field	Length (bits)	Valid Values	Additional Information
A	6	0 – 14	This field is numeric and weighted by 5 times
B	2	1 – 3	This is a repeating relation field with no adjustment. Its value is the number of times C must appear.
C	1	0 or 1	This is a choice relation field. If it is 0, fields D and E follow, if it is 1, field F follows. This field must always be 0 the first time it appears in a message.
D	1	0 or 1	
E	2	1 – 3	
F	3	0 – 2	

Table 13. A description of the fields in the example protocol

The modified Peach Pit Data Model for this protocol is shown here:

```
<DataModel name="SimpleProtocol2">
    <Number name="Field_A" size="6">
        <Hint invalid ="15-63" type="numeric" weight="5" />
    </Number>
    <Number name="Field_B" size="2">
        <Relation type="count" of="Repeating_Block" />
        <Hint valid ="1-3" />
    </Number>
    <Block name="Repeating_Block" minOccurs="0" maxOccurs="3">
        <RuleSet block="Repeating_Block" choice="Choice_Block">
            <Position token="0" present="yes" position="1" />
        </RuleSet>
        <Number name="Field_C" size="1">
            <Relation type="choice" of="Choice_Block" links="0,1" />
            <Hint valid ="all" />
        </Number>
        <Choice name="Choice_Block">
            <Block name="Choice_1">
                <Number name="Field_D" size="1">
                    <Hint valid ="all" />
                </Number>
                <Number name="Field_E" size="2">
                    <Hint invalid ="0" />
                </Number>
            </Block>
            <Block name="Choice_2">
                <Number name="Field_F" size="3">
                    <Hint invalid ="3-7" />
                </Number>
            </Block>
```

```
            </Choice>
        </Block>
    </DataModel>
```

In this appendix, coverage of the set of test cases shown in Table 14 will be computed.

Test case number	Hexadecimal	Binary
1	0F78A	00001111011110001010
2	AF4A	1010111101001010
3	0E8A	0000111010001010
4	3D7	0001000101111010
5	167B	0001011001111011
6	CD4	110011010100

Table 14. A set of test cases for the example protocol.

The coverage calculation can be divided into two major steps: finding first order coverage and finding multi-order coverage. For simplicity the calculation will use test set coverage $\mathbb{C}^2_{Fuzz\ Set}$ which is shown in (65) on page 80. This test set coverage value only relies on first and second order coverage. Therefore the first step will be to find the first order coverage of the 6 test cases (done in Section C.1), and the second step will be to find those same test case's second order coverage (done in Section C.2). Finally the two results will be combined to find $\mathbb{C}^2_{Fuzz\ Set}$ in Section C.3.

C.1 First Order Coverage

The first step in computing first order coverage of a set of test cases is to apply the first order limitations shown in Section 3.2.1 on page 65. The first limitation says that a test case may provide coverage if and only if it contains only one invalid field. To determine which test cases contain only one invalid field, they all need to be parsed and each field's validity needs to be checked. This is done in Table 15. Notice that since some of the fields repeat; values for them may appear multiple times in the same test case.

Test Case		Fields A	B	C	D	E	F	# Invalid
1	Values	3	3	0 1 1	1	3	0 2	0
	Invalid							
2	Values	43	3	0 1	1	0	2	3
	Invalid	X	X			X		
3	Values	3	2	1 1			0 2	2*
	Invalid		X	X				
4	Values	15	1	0	1	3		1
	Invalid	X						
5	Values	5	2	0 1	1	3	3	1
	Invalid						X	
6	Values	51	1	0	1	0		2
	Invalid	X				X		

* Breaks repeating choice rule, first choice must be 0

Table 15. Parsing the test cases for the example protocol

Only test cases 4 and 5 pass the first order test case limitation, so they are the only ones that will contribute coverage. The second limitation applies to fields. Fields A, B, E and F all have invalid spaces. This can be found by comparing the values considered valid with the length in bits of each field shown in Table 13. This means that fields C and D are not considered for first order coverage. The sizes for fields A, E, and F are simply the number of possible invalid values that can be entered. Since field B is a repeating relation field, its invalid space is calculated in a special way described in Section 3.1.3.1 on page 48.

With the limitations applied, the procedure listed in Section 1 on page 65 can now be followed. The first step was already achieved during the parsing, the invalid fields of test cases 4 and 5 are fields A and F respectively as shown in Table 15. The second step, sorting the test cases based on their invalid fields is trivial since there are only two test cases and two fields. Field A has one test case, test case 4, and field F has one test case, test case 5. The third step is to apply the appropriate coverage calculation for the test cases for each field.

Field A is a numeric field with no value weighting, so by Table 2, the equation to be used is (38) on page 55. Field B is not numeric, nor does it contain weighted values, so the equation to be used is (29) on page 47. Field coverage equations can require an input of the invalid space set of values and a set of test case values. The necessary inputs to the two equations and their result for this example are shown in Table 16.

Field	Equation	Inputs	Result
A	(38)	$\min(I)= 15$ $\max(I)= 63$ $\|I\| = 49$ $m = 1$ $t_i = 15$	19.85%
F	(29)	$m = 1$ $\|I\| = 5$	68.29%

Table 16. Inputs and results from applying the applicable field coverage equations.

The final step is to combine field coverage values into a singular value for first order coverage. One of the fields in this protocol, field A, is weighted. That is, the tester expects this field is 5 times as likely to contain a vulnerability than the other fields. Since field weighting is present, equation (57) on page 67 is used. If field weighting had not been present equation (58) on page 67 would have been used.

The inputs to (57) are:

- F, the number of applicable fields. This is 4, fields A, B, E, and F.
- $E[N_i]$ for $i = 1, \ldots, F$. This is the expected number of vulnerabilities. Since this figure is normalized by $E[N_{Total}]$, it is not important that the actual expected number of vulnerabilities is supplied, only that the ratio between the expectations of different fields is accurate. By default all fields receive $E[N_i] = 1$ except for the weighted field, $E[N_1] = 5$.

- C_i for $i = 1, \ldots, F$. This is the field coverage of each field. $C_1 = 19.85\%$ and $C_4 = 68.29\%$. All other $C_i = 0$ since no test cases tested those fields.

With these inputs, (57) returns a first order coverage value of $\mathbb{C}_1 = 20.94\%$.

C.2 Second Order Coverage

The first step in computing second order coverage of a set of test cases is to apply the R^{th} order limitations shown in Section 3.2.2.2 on page 71 with R = 2. The first limitation says that a test case provides second order coverage if and only if there is some combination of 2 fields that has at least one invalid value set and contains all the invalid fields and IIC pairs present in the test case. That is a very wordy statement, but it can be simplified to "the test case must have at least one invalid field but no more than two invalid fields". Unfortunately this simple language doesn't mean exactly the same thing as the limitation when complex test cases are considered, so it can not be substituted for the language in the limitation in all cases. In this case however, this statement is an equivalent limitation.

Test cases 3, 4, 5 and 6 all pass this limitation. Test case 1 fails because it does not have at least one invalid field and test case 2 fails because it has more than 2 invalid fields. This can be seen in Table 15.

The second limitation says: "A field combination is assigned a coverage value and incorporated into the final 2^{nd} order coverage metric if and only if $|I| \neq 0$." This means that it needs to be determined which field combinations in this protocol have an invalid space. This is done using (60) on page 74 reprinted here:

$$M = \binom{F}{R} - \binom{F - F_I}{R} + FC_{IIC_R} - Q_R$$

The FC_{IIC_R} term can be eliminated right away, it is 0, because the only IIC pair in

this protocol is between fields B and C because of their repeating choice relationship. Since field B has an invalid space, this combination is not included in FC_{IIC_R}. F is 6, there are 6 fields in the protocol, and F_I is 4, fields A, B, E, and F have invalid spaces. R is 2 since this is for second order field combinations. Lastly, Q_R, the number of field combinations with invalid spaces that cannot exist in the same message needs to be found.

It might be tempting to initially think that $Q_R = 2$ because the combinations of fields D and F and E and F cannot appear together if field C does not repeat. This would be correct if there was no chance for these blocks to repeat, but they can, so fields D and F and fields E and F may appear together in a message (and do in test cases 1, 2 and 5). Therefore $Q_R = 0$. Applying (60):

$$M = \binom{6}{2} - \binom{2}{2} + 0 - 0 = 15 - 1 = 14$$

There are 14 field combinations with invalid spaces. It also can be noted that there are 15 field combinations total, but the combination of fields C and D has no invalid space. This combination does not meet the limitation and is therefore dropped.

With the limitations applied, it is time to move on to applying the procedure in Section 3.2.2 on page 72. Like in the first order case,the first step was already completed during the parsing shown in Table 15. The next step involves looking at each field combination and collecting the test cases that make this combination invalid, but have no invalid fields outside of this combination. This is done in Table 17.

Next, in step 3, the field combination coverage for each field combination needs to be calculated. This is easy for combinations B–D, B–E, C–E, and D–E since they have no applicable test cases, their coverage is zero. For the remaining test cases the field coverage equation, (29) is used. While this equation was designed for field

Field Combination	Applicable test cases
A – B	4
A – C	4
A – D	4
A – E	4, 6
A – F	5
B – C	3
B – D	none
B – E	none
B – F	5
C – E	none
C – F	5
D – E	none
D – F	5
E – F	5

Table 17. Test cases that contribute to coverage of each field combination

coverage, it is used for field combination coverage as well until a better method for computing coverage over multi-order spaces is found. Since this equation assumes no relation between values, and multi-order invalid spaces can be represented as first order spaces like shown in Section 3.2.2.1 on page 68, this equation can be used with decent results. One way to improve this equation would be to include a PMF that is tailored to second order spaces. For now the generic PMF derived in Section 3.3 is used.

Equation (29) requires only the number of applicable test cases, m, and the size of the invalid space $|I|$ to compute coverage. m is trivial to obtain from Table 17, it is just the number of test cases that appear in the second column for each combination. $|I|$ is more difficult to find.

The invalid space of a second order field combination is the set of value pairs that are invalid. For a value pair to be invalid, one of the values in it must be invalid or they must be IIC. Section 3.2.2 describes these types of fields. Said more simply, the number of invalid pairs is simply the total number of pairs less the valid ones.

Most fields may take on a number of values equal to 2 raised to the number of bits in the field. The exception to this is, field B, a repeating relation field. The number of distinct values it can take on are calculated using the method in Section 3.1.3.1 on page 48. The number of value pairs between two fields is simply the product of the number of values each field can take on.

Ignoring the field combination B – C which has IIC value pairs, for any value pair to be valid, both values must be valid. The total number of valid pairs for a field combination can therefore be found by finding the product of the number of valid values in each field. The total number of pairs, minus the total number of valid pairs is logically the total number of invalid pairs, or $|I|$; Total Combined Size − Valid Combined Size = $|I|$. All this is shown in Table 18 for the field combinations without IIC value pairs.

Field Combination	Total Size			Valid Size			$\|I\|$
	Field 1	Field 2	Combined	Field 1	Field 2	Combined	
A–B	64	16	1024	15	3	45	979
A–C	64	2	128	15	2	30	98
A–D	64	2	128	15	2	30	98
A–E	64	4	256	15	3	45	211
A–F	64	8	512	15	3	45	467
B–D	12*	2	24	3	2	6	18
B–E	12*	4	48	3	3	9	39
B–F	8*	8	64	2	3	6	58
C–E	1**	4	4	1	3	3	1
C–F	1**	8	8	1	3	3	5
D–E	2	4	8	2	3	6	2
D–F	2	8	16	2	3	6	10
E–F	4	8	32	3	3	9	23

* For some number of repeats fields may not exist, or may break rules if they do exist.
This number reflects the possible values this repeating relation field can take that validly include the other field.
** For some fields to appear, this choice must be a certain value

Table 18. Calculating $|I|$ for field combinations without IIC value pairs.

To find the second order invalid space for field combination B–C, a combination of

two structure fields, the approach described in Section 3.2.3.3 on page 77 needs to be applied. Figure 43 shows the adaptation of figure 13 to this specific field combination. The size of the invalid space for field combination B–C is found to be 53.

Actual Repeats	Repeating Relation Value	Value of Field C								Number of Permutations (with replacment)	Valid	Invalid
0	0									1	0	1
0	1									1	0	1
0	2									1	0	1
0	3									1	0	1
1	0	0	1							2	0	2
1	1	0	1							2	1	1
1	2	0	1							2	0	2
1	3	0	1							2	0	2
2	0	00	01	10	11					4	0	4
2	1	00	01	10	11					4	0	4
2	2	00	01	10	11					4	2	2
2	3	00	01	10	11					4	0	4
3	0	000	001	010	011	100	101	110	111	8	0	8
3	1	000	001	010	011	100	101	110	111	8	0	8
3	2	000	001	010	011	100	101	110	111	8	0	8
3	3	000	001	010	011	100	101	110	111	8	4	4
			Valid						**Total:**	60	7	53

Figure 43. Calculation of the spaces associated with the B–C field combination

Now that $|I|$ has been found for every field combination, (29) can be applied in each case to get coverage values for all M combinations. With these coverage values, step 4 can be completed using (61). Had any field combination been weighted, the wighted version of this equation, (62), would be used instead. This returns a second order coverage value, $\mathbb{C}_2 = 41.96\%$.

C.3 Combining for Final Result

Using the simple method for arriving at coverage of a test set presented in Section 3.2.4, $\mathbb{C}^2_{Fuzz\ Set}$ can be calculated using (65). With $\mathbb{C}_1 = 20.94\%$ and $\mathbb{C}_2 = 41.96\%$, the final coverage value for this test set is $\mathbb{C}^2_{Fuzz\ Set} = 31.45\%$.

Appendix D. Open Source Protocol Vulnerability Data

A table of data on the invalid spaces of fields in a few open source protocols. This data is used to determine the PMF that describes the distributions of vulnerabilities in a variety of field types. This data and its implications are described in Section 3.3.

Table 19. Characterization of the spaces associated with some fields from select open source protocols

Protocol*	Field Name	Type	$\|I\|$	$\|V\|$	Description
GPS-SPS	Eccentricity	Numeric	4×10^9	2×10^9	Eccentricity above 0.25 causes a computational error
GPS-SPS	Reference Time	Numeric	27736	27736	Reference Time exceeds one week
GPS-SPS	Day Number	Numeric	1	1	Day number of 0 causes an error
GPS-SPS	Day Number	Numeric	248	248	Day number above 7 causes error
GPS-SPS	Satellite Configuration Code	Standard	6	6	The receiver cannot process any invalid satellite types
GPS-SPS	Spares	Standard	30	30	The receiver attempts to process the spares field because it does not contain alternating 1's and 0's
GPS-SPS	SV ID	Standard	63	43	A valid ID but on the wrong page
GPS-SPS	SV ID	Standard	63	19	An invalid ID
GPS-SPS	Special Message Character	Standard	211	32	Receiver cannot handle a control character
GPS-SPS	Special Message Character	Standard	212	21	Receiver cannot process a common, but excluded, character
GPS-SPS	Special Message Character	Standard	213	1	The ASCII delete character causes an issue

Protocol*	Field Name	Type	$\|I\|$	$\|V\|$	Description
BNEP	BNEP Type	Standard	122	122	The receiver cannot process invalid word types
BNEP	BNEP Type	Standard	122	96	The receiver is not designed to handle anything other then 0x0 or 0x7 for the first 4 bits
BNEP	UUID Size	Standard	2	1	A value of zero for message length causes a divide by zero error
BNEP	UUID Size	Standard	2	2	Any invalid value for message length causes an error
BNEP	Response Message	Standard	65531	65280	A non-zero first byte causes an issue
BNEP	List Length	Standard	65395	63488	A non-zero first four bits causes an issue
BNEP	List Length	Standard	65395	1540	Valid range, not divisible by 12
BNEP	List Length	Standard	65395	140	Divisible by 6, not 12. (contains start but not end word)
OSGP	Time Zone Offset	Numeric	65509	65487	A greater than 24 hour time zone adjustment causes an error
OSGP	Response Error Codes	Standard	244	3	Response Error codes 0x07 - 0x09 are processed since they fall between valid error codes, causing an error
OSGP	Response Error Codes	Standard	244	244	Any invalid error code causes an issue when processed

Protocol*	Field Name	Type	$\|I\|$	$\|V\|$	Description
OSGP	Integer Format	Standard	3	3	data type format other than two's compliment causes error
OSGP	Frequency	Standard	5	2	Receiver designed to reject values above the maximum of 4, but still accepts the undefined values of 0 and 1 for this field
OSGP	ID Code	Standard	212	212	Any undefined ID code causes issue
OSGP	ID Code	Standard	212	3	Undefined phase angle ID code causes issue
OSGP	Day of Week	Standard	1	1	A value of 7 is undefined (0 = Sunday, 1 = Monday, etc.)
PWL	Size	Numeric	64045	64045	Too many Payload data octets
PWL	Priority	Standard	6	6	Invalid priority
PWL	Requested Service ID	Standard	155	2	Small segment of reserved bits causes error
PWL	Service ID	Standard	155	1	Edge case undefined causes issue
PWL	Service ID	Standard	155	1	Edge case undefined causes issue
PWL	Segment Size	Numeric	256	3	Shorter than possible with require fields
PWL	Segment Size	Numeric	64079	64079	Longer than allowed by Ethernet frame
PWL	Command ID	Standard	116	25	Range between two types causes issue
PWL	Command ID	Standard	116	15	Range between two types causes issue

Protocol*	Field Name	Type	$\|I\|$	$\|V\|$	Description
PWL	Command ID	Standard	116	61	Range between two types causes issue
PWL	Command ID	Standard	116	15	Range between two types causes issue
PWL	Command ID	Standard	116	116	Any invalid type causes issue
PWL	Offset Payload data	Numeric	2	2	Too short
PWL	Offset Payload data	Numeric	64073	64073	Too long
PWL	Sub-index	Standard	65281	65281	Invalid sub-index
PWL	Sub-index	Standard	65281	1	Invalid sub-index still less than 2^8

* GPS-SPS — Global Positioning System — Standard Positioning Service

BNEP — Bluetooth Network Encapsulation Protocol

OSGP — Open Smart Grid Protocol

PWL — Ethernet POWERLINK

Bibliography

1. Euclidean algorithm. Encyclopedia of Mathematics. Available at *https://www.encyclopediaofmath.org/index.php/Euclidean_algorithm* .

2. Department of defense interface standard for digital time division command/response multiplex data bus. Technical Report MIL-STD-1553B, Department of Defense, 1978.

3. Darpa internet program protocol specification. Technical Report RFC: 791, Defense Advanced Research Projects Agency, 1981.

4. Global positioning system standard positioning service signal specification, 2nd edition. Technical report, United States Air Force, 1995.

5. D5.3 report on automated vulnerability discovery techniques. Technical Report SEC-2011.2.5-1, European Communitys Seventh Framework Programme, 2011.

6. Ieee standard for ethernet. Technical Report 802.3-2015, Institute of Electrical and Electronics Engineers, 2016.

7. Sulley. World Wide Web Page, December 2017. Available at *https://github.com/OpenRCE/sulley* .

8. D. Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc.*, 2002.

9. J. Cai, P. Zou, and D. Xiong et al. A guided fuzzing approach for security testing of network protocol software. In *2015 6th IEEE International Conference on Software Engineering and Service Science*, pages 726–729. IEEE, 2015.

10. M. Eddington. Peach fuzzing platform. World Wide Web Page, December 2017. Available at *http://peachfuzzer.com.*

11. J. Goodenough and S. Gehart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, 1975.

12. S. Gorbunov and A. Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security*, 10(8):239–245, 2010.

13. Oulu University Secure Programming Group. Protos - security testing of protocol implementations. World Wide Web Page. Available at *https://www.ee.oulu.fi/roles/ouspg/Protos.*

14. Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *2008 IEEE International Conference on Network Protocols*, pages 114–123. IEEE, October 2008.

15. C. Hu, C. Shan, W. Peng, R. Ma, and W. Ji. Fuzz testing data generation for network protocol using classification tree. In *2014 Communications Security Conference (CSC 2014)*, page 23. IET, May 2014.

16. W. Johansson, M. Svensson, U. Larson, M. Almgren, and V. Gulisano. T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 323–332. IEEE, March 2014.

17. S. Kim, W. Jo, and T. Shon. A novel vulnerability analysis approach to generate fuzzing test case in industrial control systems. In *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*, pages 566–570. IEEE, May 2016.

18. B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

19. A. Namin and J. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*, pages 57–68, New York, New York, USA, July 2009. ACM Press.

20. Microsoft Developer Network. Definition of a security vulnerability. MSDN Library. Available at *https://msdn.microsoft.com/en-us/library/cc751383.aspx*.

21. M. Sutton and A. Greene. The art of file format fuzzing. *Black Hat USA*, 2005.

22. M. Sutton, A. Greene, and P Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, Upper Saddle River, NJ, 2007.

23. E. Swihart and M. Reith. (forthcoming). redefining the air-gap for our weapon systems. In *2018 International Conference on Cyber Warfare and Security*, 2018.

24. L. Thomason. Tiny xml 2. World Wide Web Page, December 2017. Available at *https://github.com/leethomason/tinyxml2*.

25. P. Tsankov, M. Torabi Dashti, and D. Basin. Secfuzz: Fuzz-testing security protocols. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 1–7. IEEE, June 2012.

26. P. Tsankov, M. Torabi Dashti, and D. Basin. Semi-valid input coverage for fuzz testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 56–66, New York, New York, USA, July 2013. ACM.

27. E. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, 1986.

28. Michal Zalewski. mangleme. GitHub Project. Available at *https://github.com/adobe/webkit/tree/master/Tools/mangleme*.

29. H. Zhu. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
22–03–2018	Master's Thesis	Oct 2016 — Mar 2018

4. TITLE AND SUBTITLE

Expected Coverage (ExCov): A Proposal to Compute Fuzz Test Coverage within an Infinite Input Space

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Swihart Evan V., GS-11 AFLCMC/EZAC

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

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

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

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-18-M-063

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

Air Force Research Laboratory, AFMC
Attn: Lt Col Patrick Sweeney
2250 Avionics Circle
Wright-Patterson AFB, OH 45433-7765
patrick.sweeney@us.af.mil DSN: 713-4252

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYWA

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

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: PA APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A Fuzz test is an approach used to discover vulnerabilities by intentionally sending invalid inputs to a system for the purpose of triggering some type of fault or unintended effect that renders the system vulnerable to an exploit. Fuzz testing is an important cyber-testing technique used to find and fix vulnerabilities before they are exploited. The fuzzing of military data links presents a particular challenge because existing fuzzing tools cannot be easily applied to these systems. As a result, the tools and techniques used to fuzz these links vary widely in sophistication and effectiveness. Because of the infinite, or nearly infinite, number of possible fuzzed messages that can be sent on a military data link, measuring the coverage of a fuzz test is not straightforward. This thesis proposes an understandable and meaningful metric for protocol fuzz testing called ExCov. This metric computes the coverage of a fuzz test set from a probabilistic model of vulnerability occurrence and defines coverage as the expected percent of existing vulnerabilities discovered by a set of test cases. This metric enables the acquisitions community to more succinctly write weapons system requirements for cyber security. Furthermore, it quantifies the number of faults and vulnerabilities that are expected to be found by a set of test cases, which provides decision makers with valuable information to make more informed choices on whether or not to perform additional testing. As a result, industry will be better equipped to determine cost and effort when performing cyber vulnerability testing. In addition, industry will also be able to more concretely represent the results of the cyber testing they perform. ExCov was implemented in a suite of tools called ExFuzz, and these tools were used to compare and contrast military data link fuzz testing techniques that are in use today. By assessing these current methods using the ExCov metric, optimal bit flip probabilities for the mutative fuzzing of three custom protocols was found. A generative fuzzer was also built based on the metric and was shown to outperform mutative and manual generation strategies in nearly every case.

15. SUBJECT TERMS

Fuzzing, Coverage Criterion, Military Data Links, Vulnerability Discovery

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Timothy J. Carbino, AFIT/ENG
U	U	U	U	182	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x4220; timothy.carbino@afit.edu