Dartmouth College
Dartmouth Digital Commons

Dartmouth College Ph.D Dissertations | Theses and Dissertations

3-1-2013

Security-Policy Analysis with eXtended Unix Tools

Gabriel A. Weaver
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation

Weaver, Gabriel A., "Security-Policy Analysis with eXtended Unix Tools" (2013). *Dartmouth College Ph.D Dissertations*. 38.
https://digitalcommons.dartmouth.edu/dissertations/38

This Thesis (Ph.D.) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Ph.D Dissertations by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Security-Policy Analysis with eXtended Unix Tools

Dartmouth Computer Science Technical Report TR2013-728

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in
Computer Science
by

Gabriel A. Weaver
DARTMOUTH COLLEGE
Hanover, New Hampshire

March, 2013

Examining Committee:

____

(chair) Sean W. Smith, Ph.D.

____

Rakesh B. Bobba, Ph.D.

____

Thomas H. Cormen, Ph.D.

____

M. Douglas McIlroy, Ph.D.

____

Daniel Rockmore, Ph.D.

____

F. Jon Kull, Ph.D.
Dean of Graduate Studies

Abstract

During our fieldwork with real-world organizations—including those in Public Key Infrastructure (PKI), network configuration management, and the electrical power grid—we repeatedly noticed that security policies and related security artifacts are hard to manage. We observed three core limitations of security policy analysis that contribute to this difficulty. First, there is a gap between policy languages and the tools available to practitioners. Traditional Unix text-processing tools are useful, but practitioners cannot use these tools to operate on the high-level languages in which security policies are expressed and implemented. Second, practitioners cannot process policy at multiple levels of abstraction but they need this capability because many high-level languages encode hierarchical object models. Finally, practitioners need feedback to be able to measure how security policies and policy artifacts that implement those policies change over time.

We designed and built our eXtended Unix tools (XUTools) to address these limitations of security policy analysis. First, our XUTools operate upon context-free languages so that they can operate upon the hierarchical object models of high-level policy languages. Second, our XUTools operate on parse trees so that practitioners can process and analyze texts at multiple levels of abstraction. Finally, our XUTools enable new computational experiments on multi-versioned structured texts and our tools allow practitioners to measure security policies and how they change over time. Just as programmers use high-level languages to program more efficiently, so can practitioners use these tools to analyze texts relative to a high-level language.

Throughout the historical transmission of text, people have identified meaningful substrings of text and categorized them into groups such as sentences, pages, lines, function blocks, and books to name a few. Our research interprets these useful structures as different context-free languages by which we can analyze text. XUTools are already in demand by practitioners in a variety of domains and articles on our research have been featured in various news outlets that include ComputerWorld, CIO Magazine, Communications of the ACM, and Slashdot.

Acknowledgments

Although the rewards are great, the pursuit of a Ph.D is a perilous path fraught with untold dangers. While I cannot possibly enumerate everyone who has prepared and helped me on this journey, I do want to thank my family, friends, colleagues, and funders.

I could not have possibly gotten to this point without the support of my family. I want to thank my father for teaching me about hard work and discipline through our nightly father and son games of catch and gardening sessions. I want to thank my mom for reading to me continuously from the time I was born and for cooking such excellent food. I want to thank my brother and sister, Mike and Peach for taking care of their little brother through all of the emotional rollercoasters of the experience and for reminding me that there was more to life than school.

I've been blessed with many friends through the years so rather than listing them all, I want to list a few of the people that have impacted my life the most the past few years. First, I want to thank Jim and his wife Abby. Jim is a steadfast man of exemplary character and has supported me from the moment I moved out of our apartment at 433 Cambridge Street. He helped me move, visited me, and kept me sane when I reached my breaking point (on several occasions). He has found his equal in his wife, Abby (a.k.a. Dr. Wow). I want to thank Alison Babeu and her husband Alex for their dry, witty sense of humor, for keeping me in line, and for driving 2.5 hours from Boston just to walk their dog and treat me to dinner. They are steadfast friends and make a killer mexican dinner. I want to thank Kate MacInnes for introducing me to running distances all over the hills of New Hampshire and Vermont. I'll always remember our great runs. I want to thank Nick Foti. Nick and I both took classes at Tufts and we started graduate school at Dartmouth in the same year. He's a serious mathematician and a great person with whom to eat lunch or walk around Occum Pond. Finally, I want to thank Janna Genereaux and her family for welcoming me into their home during the final few months of graduate school. I'll always remember their generosity, kindness, and warmth.

If I live to be 90 I will have spent one-third of my life in school. Therefore, I would be remiss if I didn't thank the teachers who prepared me for graduate school. I want to thank my Pre-K teacher Mrs. Manso at St. James for creating Science Discovery Camp. In addition to providing me with a captive lunchtime audience for my harmonica playing, science discovery camp taught me the value of hands-on science experiments. I want to thank Ernest Dodge, my high-school English and Latin

teacher for introducing me to the value of a Classical education and to value logic, reason, and memorization. It is because of Mr. Dodge that I was able to navigate my father's hospitalization with some sanity. I want to thank Neel Smith, an archaeologist and Professor of Classics at Holy Cross. When everyone else thought I was crazy, Neel encouraged me to combine my love of the Classics and Mathematics and allowed me to do an independent study on the diagrams of Euclid and Theodosius. Later on, he, along with Harvard's Center for Hellenic Studies, allowed me to pursue a once-in-a-lifetime opportunity: to digitize diagrams for the Archimedes Palimpsest Project. Finally, I want to thank Greg Crane, a Professor of Classics at Tufts University, for allowing me to continue my research in the domain of Classics and Mathematics for three years after graduating from Holy Cross.

During my time at Dartmouth, I was fortunate enough to work with a great set of colleagues. I want to thank Sergey and Anya, who found me without an advisor and suggested that I think about security and its ties to natural-language processing. Once I was in the PKI/Trust lab, I met many great people including Scout, Bx, Chrisil, and Jason that could empathize with the plight of the Ph.D student. I want to thank Suzanne Thompson, a woman who tirelessly worked to provide a high-quality summer camps for students interested in computers and society. She is a pleasure to work with and a fabulous person. I also want to thank colleagues at the Institute for Security, Technology, and Society. In particular, I want to thank Tom Candon, who offered countless words of encouragement and handfuls of candy.

My thesis research was only possible because of the support of domain experts. I want to thank Scott Rea for teaching me about the value of large families, character, and planning a trip to Europe before actually going. I will always remember our adventure in the Milan train station and its many armored police vehicles and shady characters. I also want to thank Edmond Rogers for his direction on network configuration management in the power grid and his continuing support and friendship since I have moved to Illinois. Although extremely loud, his actions speak even louder and are a testament to his character.

I want to thank my thesis committee for helping me to pursue an unconventional thesis topic. Thanks to Rakesh Bobba for agreeing to be on my thesis committee and continuing to give me guidance at Illinois. Thanks to Tom Cormen for insisting upon a more formal foundation for my tools at my proposal. I will always remember Tom's stories about hockey, baseball, and barbeque. Thanks to Doug McIlroy, a true Unix god, for his countless suggestions and continued enthusiasm for XUTools. Thanks to Dan Rockmore for his interest in applying XUTools to data mining and legislation

even when few others saw their potential.

Most importantly, I want to thank my advisor Sean W. Smith. Sean is a man of character and he stood by me through thick and thin. Even when my research ideas were shaky and my future uncertain, he provided guidance to see me through. I can't thank Sean enough for funding my research, especially when others questioned whether it was Computer Science. It is difficult to state just how much Sean has positively impacted my life and he continues to do so even after my defense.

Finally, I want to thank my funding sources for making my research possible. The National Science Foundation, Google, and the Department of Energy all made this thesis possible. The views and conclusions of this thesis do not necessarily represent the views of the sponsors.

Contents

1 Introduction **1**
- 1.1 Problem Scenarios 2
  - 1.1.1 Real-World Security Domains 2
  - 1.1.2 Limitations of Security Policy Analysis 5
- 1.2 This Thesis 7
  - 1.2.1 Theory for Structured Text 7
  - 1.2.2 Why and How to Use XUTools 7
  - 1.2.3 Design and Implementation of XUTools 8
  - 1.2.4 Evaluation 8
  - 1.2.5 Application 9
  - 1.2.6 Future Work 9
  - 1.2.7 Conclusions and Appendices 10

2 Problem Scenarios **11**
- 2.1 X.509 Public Key Infrastructure Policies 12
  - 2.1.1 Policy Comparison 12
  - 2.1.2 Drawbacks of Current Approaches 15
  - 2.1.3 Limitations of PKI Policy Analysis 16
  - 2.1.4 Section Summary 19
- 2.2 Network Configuration Management 19
  - 2.2.1 Summarize and Measure Change 20
  - 2.2.2 Drawbacks of Current Approaches 21
  - 2.2.3 Limitations of Summarizing and Measuring Change 24
  - 2.2.4 Section Summary 26
- 2.3 Electrical Power Grid 26
  - 2.3.1 Change Control and Baseline Configuration Development 28
  - 2.3.2 Drawbacks of Current Approaches 29

2.3.3 Limitations of Change Control and Baseline-Configuration Development . . . 30
2.3.4 Section Summary . . . 34
2.4 Conclusions . . . 34

3 Theory **35**
3.1 Theoretical Background . . . 36
3.1.1 Language Theory . . . 36
3.1.2 Parsing . . . 45
3.1.3 Security Policy Corpora as Data Types . . . 47
3.1.4 Section Summary . . . 52
3.2 How We Address Limitations of Security-Policy Analysis . . . 52
3.2.1 Policy Gap Problem . . . 52
3.2.2 Granularity of Reference Problem . . . 54
3.2.3 Policy Discovery Needs Problem . . . 56
3.3 Conclusions . . . 57

4 Why and How to Use XUTools **58**
4.1 XUTools and Real-World Use Cases . . . 58
4.1.1 XUGrep . . . 58
4.1.2 XUWc . . . 65
4.1.3 XUDiff . . . 69
4.2 Conclusions . . . 75

5 Design and Implementation of XUTools **76**
5.1 Design Requirements . . . 76
5.1.1 Tools Gap Problem . . . 77
5.1.2 Granularity of Reference Problem . . . 78
5.1.3 Policy Discovery Needs Problem . . . 78
5.2 XUTools Internals . . . 79
5.2.1 XUGrep Internals . . . 79
5.2.2 XUWc Internals . . . 85
5.2.3 XUDiff Internals . . . 88
5.2.4 Grammar Library . . . 90
5.2.5 xupath . . . 92
5.3 Conclusions . . . 98

6 General Evaluation of XUTools 99
6.1 XUGrep . . . 99
6.1.1 Evaluation—Qualitative . . . 99
6.1.2 Evaluation—Quantitative . . . 100
6.1.3 Related Work . . . 102
6.2 XUWc . . . 103
6.2.1 Evaluation—Qualitative . . . 104
6.2.2 Evaluation—Quantitative . . . 104
6.2.3 Related Work . . . 105
6.3 XUDiff . . . 106
6.3.1 Evaluation—Qualitative . . . 106
6.3.2 Evaluation—Quantitative . . . 107
6.3.3 Related Work . . . 107
6.4 Grammar Library . . . 108
6.5 Conclusions . . . 109

7 Application of XUTools to Network Configuration Management 110
7.1 Introduction . . . 110
7.2 XUTools Capabilities for Network Configuration Management . . . 112
7.2.1 Inventory of Network Security Primitives . . . 112
7.2.2 Similarity of Network Security Primitives . . . 116
7.2.3 Usage of Network Security Primitives . . . 123
7.2.4 Evolution of Network Security Primitives . . . 128
7.2.5 Section Summary . . . 130
7.3 Evaluation . . . 131
7.3.1 General Feedback from Practitioners . . . 132
7.3.2 Related Work . . . 134
7.3.3 Capability-Specific Evaluation . . . 136
7.4 Conclusions . . . 140

8 Future Work 141
8.1 Ongoing Research . . . 141
8.1.1 Application of XUTools to X.509 PKI . . . 141
8.1.2 Application of XUTools to Terms of Service Policies . . . 144
8.1.3 Application of XUTools to the Power Grid Data Avalanche . . . 146
8.2 Additional Problem Scenarios . . . 149

8.2.1 Healthcare Information Technology 149
8.2.2 Legislation and Litigation 153
8.2.3 Operating Systems and Trusted Hardware 154
8.3 Additional XUTools Extensions 155
8.3.1 Version Control 155
8.3.2 Grammar Library 156
8.3.3 Distributed Parsing 156
8.3.4 Distance Metrics for XUDiff 156
8.3.5 Current XUTools: 158
8.3.6 New XUTools 158
8.3.7 A GUI for XUTools 159
8.4 Conclusions 161

9 Conclusions 162

A Pre-XUTools PKI Policy Analysis Tools 164
A.1 PKI Policy Repository 164
A.1.1 Security Policy Analysis Problems Addressed 165
A.1.2 Design and Implementation 167
A.1.3 Evaluation 167
A.2 Policy Builder 168
A.2.1 Security Policy Analysis Problems Addressed 168
A.2.2 Design and Implementation 169
A.2.3 Evaluation 169
A.3 Policy Mapper 170
A.3.1 Security Policy Analysis Problems Addressed 170
A.3.2 Design and Implementation 170
A.3.3 Evaluation 170
A.4 Vertical Variance Reporter 172
A.4.1 Security Policy Analysis Problems Addressed 172
A.4.2 Design and Implementation 172
A.4.3 Evaluation 173
A.5 Policy Encoding Toolchain 173
A.6 Conclusions 174

B PyParsing Internals 175
B.1 PyParsing and Recursive Descent Parsers 176

B.1.1 Interpretation Functions 176
B.1.2 Parser Combinators 179
B.1.3 Combinators and Recursive-Descent Parsers 183
B.2 Evaluation of Recursive Descent Parsers 186
B.2.1 Evaluation—Implementation Complexity 186
B.2.2 Evaluation—Recognition Power 189
B.2.3 Evaluation—Usability and Portability 189
B.2.4 Alternative Parsing Algorithms 190
B.3 Implementation and Evaluation of Scan String 191
B.4 Conclusions 193

List of Tables

5.1 Sizes of grammars used by our XUTools 92
5.2 Types of nodes in a xupath parse tree 97

7.1 An inventory of object groups and access-control lists across the Dartmouth network 116
7.2 Similarity of ACLs in the Dartmouth Network 121
7.3 Evolution of number of object groups in Dartmouth network 129
7.4 Evolution of number of ACLs in Dartmouth network 130
7.5 Evolution of number of ACLs used in Dartmouth network 131

8.1 Multiversioned corpus of EUGridPMA security policies 143
8.2 Multiversioned corpus of terms of service and privacy policies 146

B.1 Examples of tokens for TEI-XML 178
B.2 Example of a parser for a TEI-XML *head* element 185

List of Figures

3.1 Definition of *string* 37
3.2 Definition of *language* 37
3.3 Definition of *recognizer* 39
3.4 Classes of languages 40
3.5 Definition of *context-free grammar* and *parser* 43
3.6 Definition of *parse tree* 46
3.7 Equivalence classes of strings 48
3.8 Parsing as an operation on strings 49
3.9 String and tree edit distance 51
3.10 Language classes of XUTools 53
3.11 Levels of abstraction of a parse tree 55
3.12 Security policy change trends 56

4.1 Usage of `xugrep` 59
4.2 Example Cisco IOS configuration file 60
4.3 Using `xugrep` to process network configuration files 61
4.4 Example C source files 62
4.5 Using `xugrep` to process C source files 63
4.6 Usage of `xuwc` 66
4.7 Example of multi-versioned Cisco IOS configuration file 67
4.8 Using `xuwc` to process network configuration files 68
4.9 Usage of `xudiff` 70
4.10 Using `xudiff` to compare network configuration files 72
4.11 Using `xudiff` to compare network configuration files with varying cost functions 73

5.1 Step 1 of how `xugrep` extracts lines per network interface 80
5.2 Step 2 of how `xugrep` extracts lines per network interface 81
5.3 Step 3 of how `xugrep` extracts lines per network interface 81

5.4 How xugrep reports matches at multiple levels of abstraction 82
5.5 xugrep algorithm internals 83
5.6 xugrep algorithm implementation 84
5.7 Step 1 of how xuwc counts lines per network interface 86
5.8 Step 2 of how xuwc counts lines per network interface 86
5.9 Step 3 of how xuwc counts lines per network interface 87
5.10 How xudiff compares two network device configurations 89
5.11 Example Apache server configuration file 94
5.12 Implementation of xupath parse tree 96

6.1 Using xugrep to process NVD-XML 101

7.1 Using XUTools to inventory network security primitives 115
7.2 Step 1 of first method to compare similarity of ACLs 117
7.3 Step 2 of first method to compare similarity of ACLs 118
7.4 Similarity of Object Groups in the Dartmouth Network 122
7.5 Detailed view of Object Group cluster 123
7.6 Using xudiff to view clustered Object Group differences 124
7.7 Using xuwc to count total number of ACLs in the Dartmouth core network 125
7.8 A pipeline to count number of unique ACLs applied to a network interface 127

A.1 Semantic gap within PKI policy audit 166
A.2 Our PKI Policy Repository 166

B.1 Defininition of *interpretation function* 177
B.2 The *Or* combinator 180
B.3 A PyParsing grammar for security policies encoded in TEI-XML . . . 181
B.5 The *And* combinator 181
B.4 Fragment of a CPS encoded in TEI-XML 182
B.6 Call graph of an grammar written with parser combinators 184
B.7 An alternative TEI-XML grammar to demonstrate backtracking . . . 187
B.8 Part 1 of an example to demonstrate backtracking in recursive-descent parsers 187
B.9 Part 2 of an example to demonstrate backtracking in recursive-descent parsers 188

Chapter 1

Introduction

Security policies are fundamental to the traditional notion of security. Traditional Orange-Book security methods formalize *Discretionary Access Control (DAC)* and *Mandatory Access Control (MAC)* policies with lattices and matrices [76]. In the real-world, however, practitioners work with a broader definition of security policy—a set of rules designed to keep a system in a good state [120]. The terms *rules*, *system* and *good* mean different things in different domains. For example, in an enterprise, the rules may take the form of a natural-language legal document designed to ensure the availability and correctness of an authentication system. Alternatively, the rules of a security policy might be expressed as a configuration file designed to ensure proper access to network resources.

Researchers have placed less emphasis, however, on how to help humans produce and maintain security policies. In addition, practitioners must also produce and maintain security policy artifacts such as configuration files and logs, that implement and reflect security posture.

1.1 Problem Scenarios

During our fieldwork, we observed that the term *security policy* means different things to practitioners within different domains. We describe our fieldwork in three real-world security domains in Chapter 2. We define what the term *security policy* means within the context of each domain. Current approaches to analyze security policies and related policy artifacts suffer from several drawbacks and this makes policy management inefficient, inconsistent, and difficult in general. These drawbacks are symptoms of three core limitations of security policy analysis that repeatedly appear in a variety of domains.

- *Policy Gap Problem*: There is a capability gap between traditional text-processing tools (e.g. `grep`,`diff`) and the policy languages we encountered during our fieldwork.
- *Granularity of Reference Problem*: Practitioners need to be able to process texts on multiple levels of abstraction and currently they cannot.
- *Policy Discovery Needs Problem*: Practitioners need feedback so that they can measure properties of security policies and how they change.

We now introduce three security domains in which we observed these limitations of security-policy analysis.

1.1.1 Real-World Security Domains

X.509 Public Key Infrastructure (PKI):

In the domain of X.509 *Public Key Infrastructure (PKI)*, practitioners specify and implement security policies via *Certificate Policies (CPs)* and *Certification Practices Statements (CPSs)* respectively. These natural-language legal documents govern how

practitioners create, maintain, and revoke *PKI certificates*—digital documents that encode associations between a public key and one or more attributes. The Internet Engineering Task Force (IETF) Request for Comment (RFC) 2527 and 3647 define a standard format for these policies [27, 28].

Failure to properly manage CPs and CPSs has real consequences that may result in improper access, for example to classified U.S. Federal government facilities or information. X.509 is the basis for *Personal Identity Verification (PIV)* cards that authorize physical access to Federal facilities including the Nuclear Regulatory Commission and the State Department. The *level of assurance* associated with a certificate depends upon the contents of the security policy. Mismanagement of policies can have serious consequences. For example if a commerical CA such as DigiCert produced a CP/CPS that adhered to the *CA/Browser (CAB)* Forum's *Baseline Requirements*, then that policy would be a pivotal attribute to ensure that it was included in popular browsers and operating systems [15]. If, however, the commercial CA updated their CP/CPS so that they were no longer compliant that level of assurance, then they would be rejected from those browsers and operating systems, and this would dissuade anyone from trusting them.[1]

Network Configuration Management

For a network administrator or auditor, the term *security policy* refers to a configuration file for a network device. In the taxonomy of our broader research, however, we view these files as security policy artifacts because they implement rules given by a high-level security policy. These high-level policies are written to satisfy compliance standards such as ISO/IEC 27001 [66] or NERC CIP [88]. Network administrators must routinely change the configuration of their network to accomodate new

[1] Scott Rea called this scenario "business suicide" for a commercial CA. Scott Rea was the Senior PKI Architect for Dartmouth College and he is currently a Senior PKI Architect at DigiCert. He also was a founding member and is the current Vice Chair of *The Americas Grid Policy Management Authority (TAGPMA)* [104]

services and keep the network secure and compliant with regulations. If network administrators don't update their policies, then their networks are vulnerable to new threats. Network operators consider network configuration files to be the "most accurate source of records of changes" [125, 126].

Failure to properly manage network configurations has significant consequences, which include network outages. When a network administrator updates a security policy and changes network device configurations, he can introduce misconfigurations. Misconfigurations cause most network outages according to a study by Oppenheimer [92]. For example, a major outage of *Amazon's Elastic Compute Cloud (Amazon EC2)* in July 2011 was caused by operator misconfiguration [13].

Electrical Power Grid

In the domain of the electrical power grid, the term *security policy* may refer to *North American Electric Reliability Corporation's Critical Infrastructure Protection (NERC CIP)* regulatory guidelines or to a configuration file for an *Intelligent Electronic Device (IED)*. One example of an IED is a *relay*, a device that protects electrical equipment from overload via a breaker. Newer relays are controlled via an ethernet-based protocol and so must be configured correctly or else expensive equipment can be damaged.

Power system control networks must comply with NERC CIP regulations. The consequences of failing to fulfill these provisions are severe. According to one industry expert who has performed audits at a major utility, fines scale up to 1.5 million dollars per day of violation retroactive to the beginning of the offense.

As the smart grid grows, more devices will be deployed to sense and control the state of the electrical power grid. Another kind of device on a power control network is a *Remote Terminal Unit (RTU)*. RTUs present information to operators. The networks upon which these devices sit, their access-control policies, and the

logs they generate all govern the security posture of the organization involved. Poor configuration may lead to cyber-physical attacks or power outages. Some say that civilization as we know it will last 14 at most days without electrical power.[2]

1.1.2 Limitations of Security Policy Analysis

The Policy Gap Problem, Granularity of Reference Problem, and Policy Discovery Needs Problem make security policies hard to manage in each of the domains mentioned above.

Policy Gap Problem

There is a gap between the tools available to practitioners and the languages practitioners use to represent security policies and security policy artifacts. For example, in X.509 PKI, a *Certificate Authority (CA)* creates, maintains, and revokes certificates. An experienced enterprise CA officer estimated that it takes him 80–120 hours to compare two Certificate Policy (CP) or Certification Practices Statement (CPS) documents.[3] Policy comparision takes a week or more of man hours because policy analysts operate on PKI policies by their reference structure (defined in RFC 2527 or RFC 3647). Other representations of policy such as PDF, however, are organized by page. The page-centric organization of PDF viewers, combined with the complexity of parsing the PDF format [113, 144] imposes a semantic gap that forces policy operations to be largely manual. Consequently, analysts must manually translate, in their heads, between policy page numbers and a reference structure.

[2]Conversations with Edmond Rogers. Edmond Rogers used to secure the power-control network of a major *Investor Owned Utility (IOU)*. He currently serves as the utility expert on the DOE-funded TCIPG project.

[3]Conversations with Scott Rea.

Granularity of Reference Problem

Practitioners need to be able to process policies at multiple levels of abstraction and currently cannot. For example, network configurations are expressed at many different layers of abstraction. Network configuration files may be grouped according to network topology (core, wireless, etc), routers may define various virtual *Local Area Networks (LANs)*, and even router configurations themselves are written in a hierarchical language. Unfortunately, current tools do not allow practitioners to process policies or their artifacts at arbitrary levels of abstraction.

Policy Discovery Needs Problem

Practitioners and security researchers need feedback to understand trends in how security policies change over time. Security systems are deployed within dynamic environments. New vulnerabilities, new features, and new technologies are all reasons why policies must evolve. In software engineering, Lehman denotes such a system as *e-type*: a system embedded in the real-world and whose correctness depends upon the "usability and relevance of its output in a changing world" [77].

If practitioners don't update their security policies, then their systems are vulnerable to attack. If practitioners do update their security policies, however, they may introduce misconfigurations.

A recent conversation with an auditor of power control systems revealed that many utilities and auditors have no way to even know what *normal* is when it comes to the configuration of control-system networks or devices on those networks.[4] Feedback on how security policies change over time would help auditors and administrators understand what normal looks like and how their security posture changes.

[4] Conversation with Bryan Fite at TCIPG Industry Day, 2013. Bryan Fite is a Security Portfolio Manager at *British Telecommunications (BT)*. In addition, he runs the DAY-CON security summit [37] as well as PacketWars, an information warfare simulation [93].

1.2 This Thesis

We now discuss the organization of the remainder of this thesis. Each subsection corresponds to a different thesis chapter.

1.2.1 Theory for Structured Text

During our fieldwork, we observed that in general, many of the policies and associated artifacts that we encountered—whether expressed in markup, configuration, or programming languages—were structured texts. Our approach to security policy analysis relies upon our observation that we can leverage formal language theory to analyze a wide variety of security policies. We will review formal language theory and other theoretical components of our approach in Chapter 3. Later, we apply this theoretical toolbox to directly address our three limitations of security policy analysis.

First, we hypothesize that the gap between policy languages and tools available to practitioners exists because traditional text-processing tools compute on regular languages that cannot recognize the hierarchical object models in which security policies and their artifacts often are written and represented.

Second, we observe that context-free grammars recognize languages of strings with arbitrarily deep hierarchies and that parse trees for these strings provide a natural formalism to process multiple layers of abstraction.

Finally, we use our theoretical toolbox to enable practitioners to measure how security policies change over time.

1.2.2 Why and How to Use XUTools

Our *eXtended Unix Tools (XUTools)* enable practitioners to extract (`xugrep`), count (`xuwc`), and compare (`xudiff`) texts in terms of their hierarchical syntactic structure.

In Chapter 4, for each of these tools, we describe their usage, motivate their design with a selection of use cases, and provide specific examples of how to use each tool to address a usage scenario. Some of these use cases come directly from the three, aforementioned security domains. Other uses for our tools however, come from system administratrs, auditors, and developers who became interested in our tools after our LISA 2011 poster was featured on Slashdot [107].

1.2.3 Design and Implementation of XUTools

Our XUTools address the three core limitations of security policy analysis. Chapter 5 explains the design and implementation of XUTools in more detail. First, we explain how we designed our tools to directly address the Policy Gap Problem, Granularity of Reference Problem, and Policy Discovery Needs Problem. We then describe the implementation of each of our XUTools; we provide a detailed example of how our tool works, a description of the tool's underlying algorithm, and a discussion of its implementation.

1.2.4 Evaluation

We evaluated our XUTools quantitatively and qualitatively. The quantitative evaluation includes the worst-case time complexity of our tools, implementation details such as lines of code and test coverage, and the performance of our tools. The qualitative evaluation consists of anecdotal feedback on our tools from real-world practitioners and a discussion of how our research improves upon current approaches. More information about our evaluation may be found in Chapter 6.

1.2.5 Application

We applied XUTools so that network administrators and auditors can answer practical questions about the evolving security posture of their network. Our XUTools-enabled capabilities directly address the problem scenarios discussed in Chapter 2 and demonstrate how we can use our toolset to formalize real-world textual analyses used by practitioners.

Specifically, real-world network administrators on college campuses as well as auditors of electrical power control networks require the ability to summarize and measure changes to router configuration files. Longitudinal analyses of network configuration files can help identify misconfigurations (a source of outages) and provide evidence for compliance during audit. We used our XUTools to quantify the evolution of network security primitives on the Dartmouth College campus from 2005-2009. Practitioners can create an inventory of network security primitives, measure the similarity between objects in that inventory, see how those primitives are used, and then measure the evolution of the network through time. As a result, network administrators and auditors alike can use our tools to analyze how a network changes at a variety of levels of abstraction. More information may be found in Chapter 7.

1.2.6 Future Work

We can extend our work in several directions. Chapter 8 proposes new problem scenarios, extensions to our XUTools that also extend our theoretical toolbox, and pilot studies that apply XUTools to enterprise security policies and to other aspects of the smart grid.

1.2.7 Conclusions and Appendices

In Chapter 9 we conclude. At the back of the thesis we provide several appendices. Appendix A motivates, describes, and evaluates our non-XUTools based solutions to the three core limitations of policy analysis in the context of PKI. In Appendix B, we provide additional details about the PyParsing library that our XUTools uses to define recursive-descent parsers.

Chapter 2

Problem Scenarios

The term *security policy* means different things to practitioners within different domains. In this chapter, we describe observations from our fieldwork in three real-world security domains. For each domain, we define what the term *security policy* means. We observed that policy management is inefficient, inconsistent, and generally difficult. These policy management issues are symptoms of three core limitations of security policy analysis that repeatedly manifest themselves in a variety of domains. Recall our three core limitations introduced in Chapter 1.

- *Policy Gap Problem*: There is a gap between the tools available to practitioners and the languages practitioners use to represent security policies and security policy artifacts.

- *Granularity of Reference Problem*: Practitioners need to be able to process policies at multiple levels of abstraction and currently they cannot.

- *Policy Discovery Needs Problem*: Practitioners and security researchers need feedback so that they can measure properties of security policies and how they change.

2.1 X.509 Public Key Infrastructure Policies

X.509 PKI certificates are created, maintained, and revoked by a Certificate Authority (CA), who attests to the validity of associations between a public key and one or more attributes. When these attributes serve to identify a person, machine, or organization, certificates may be used to authenticate a user to a computer system or even to grant a person physical access to facilities.

In X.509 PKI the term *security policy* refers to Certificate Policies (CPs) and Certification Practices Statements (CPSs) respectively. An organization's CP contains the set of expectations that define that organization's notion of a trustworthy public key certificate, the certificate's level of assurance, and how that certificate may be used. The CPS states how a CA actually implements a CP. RFC 2527 and RFC 3647 define a framework to facilitate policy creation and comparison [27, 28].

Our Fieldwork: Throughout our research on X.509 PKI, we worked closely with Scott Rea, the former Senior PKI Architect for Dartmouth College and a Senior PKI Architect at DigiCert. Through Scott, we were able to observe PKI audits and present our work at meetings of the *Federal PKI Policy Authority (FPKIPA)*, the *European Union Grid Policy Management Authority (EuGridPMA)*, and *The Americas Grid Policy Management Authority (TAGPMA)*.

2.1.1 Policy Comparison

For CAs and policy analysts, policy comparision is an important part of three X.509 processes and failing to perform these processes well has serious consequences.

PKI Compliance Audit: PKI compliance audits verify that an organization's CPs and CPSs are consistent with a baseline framework of requirements via policy comparison. Policy comparison requires the analyst to compare sections of one policy

with the corresponding sections in another. In theory, these sections should match, but in practice often they do not and they may be moved or missing. Policy comparision is required to map these high-level compliance criteria to the actual CP and CPS documents.

The importance of compliance audit is recognized across a variety of industries. In the financial services industry, compliance audits evaluate PKIs with respect to security, availability, processing integrity, confidentiality, and privacy. ISO 21188 specifies a framework that evolved from WebTrust and ANSI X9.79 [73]. Audits for WebTrust compliance should occur every 6 months [52].

PKI compliance audits ensure that actual observed practices are consistent with the practices stated in a CPS. Failure to pass a compliance audit may result in regulatory fines and result in a loss of the ability to service restricted or closed markets (such as the *Department of Defense (DoD)* contract market).

IGTF Accreditation: Researchers that use computational grids employ many thousands of distributed nodes to solve complex computational problems by sharing resources. Since these resources are valuable, access is limited based on the requested resource and the user's identity. Each grid provides secure authentication of both its users and its applications in order to enforce resource limits [94].

The *International Grid Trust Federation (IGTF)* develops standards for certificates used to authentication to e-Science production infrastructures. The IGTF accreditation process compares a member CA's security policy against a baseline *Authentication Profile (AP)*. An AP specifies legal and technical requirements for certificates used to authenticate to an e-Science grid.

The IGTF accreditation process is important because it ensures the consistency of authentication standards across a distributed architecture (e-Science grids). To implement the accreditation process, the IGTF pairs prospective member CAs with

volunteer member CAs. During accreditation, the prospective member CA sends a draft CP to other members for comments and asks the volunteer members to review the policy in detail. The prospective member CA eventually presents this CP, along with recommendations from reviewers, at a meeting for the IGTF to approve or defer immediately.

If the IGTF accredits a non-compliant organization or denies accreditation to someone who is compliant, the consequences are severe. In the former case, if non-compliance at the time of accreditation is exposed, the IGTF as a whole is exposed to legal risk. In the latter case, if the IGTF bans a noncompliant organization, that organization's researchers lose access to high-performance grid computing resources that may be vital to research.

Policy Mapping to Bridge PKIs: Bridge CAs, though not themselves anchors of trust, establish relationships with different PKIs by comparing their policies against a set of baseline requirements. Once a baseline relationship has been established, users from different PKIs can decide whether to accept one another's certificates.

Bridges exist to mediate trust in several areas that include the pharmaceutical industry (through *Signatures and Authentication for Everyone-BioPharma (SAFE-BioPharma)*), the U.S. Federal government (through the *Federal PKI Policy Authority (FPKIPA)*), the aerospace and defense industry (CertiPath), and higher education (through the *Research and Higher Education Bridge CA (REBCA)*) [1].

In order to create PKI bridges, the bridge CA must map policies between member PKIs. When a new organization wishes to join a bridge, the bridge CA compares the candidate organization's CP to its own. If suitable, the bridge CA will sign the certificate of the candidate organization's trust root.

2.1.2 Drawbacks of Current Approaches

The process by which CAs and analysts compare policies makes it expensive for grids, bridges, and organizations to review processes in a timely and frequent manner. Although policy comparison is a fundamental operation required to create bridge PKIs and to accredit grid member organizations, it remans a manual, subjective process, making it inefficient and difficult to perform consistently. To evaluate the similarity of two CPs, CAs compare the policies line-by-line. For a person with extensive experience, this can take 80–120 hours depending upon whether the two policies were written according to the same IETF standard. Compliance audits, accreditation procedures, and policy mapping decisions are difficult to reproduce because they are highly dependent upon auditors' individual observations. Were an auditor to try to reproduce an audit, the conditions under which the original audit occurred might be extremely difficult or impossible to reproduce.

Even if the data for these X.509 processes were reproducible, the data would only capture the state of the organization at a single point in time. Organizations are dynamic, changing entities. Audits rely upon the past as the sole indicator of current and future performance.

Within the identity management literature, researchers have proposed several different approaches that address aspects of the policy comparision problem. We will now discuss relevant research in policy formalization, retrieval, creation, and evaluation.

SAML [17] and XACML [86] formalize authentication and authorization policies in XML. Chadwick et al. developed various XML-based Role-Based Access Control (RBAC) authorization policies so that domain administrators and users can manage their own resources [21, 65].

Previous work in certificate policy formalization focuses less on human-readable, machine-actionable representations of policy. Blaze et al. [10], Mendes et al. [83],

and Grimm et al. [57] all use ASN.1 to model properties inferred from the policy's source text. Others such as Casola et al. [18, 19], have developed data-centric XML representations, suitable for machines. These representations, however, are not suitable because human analysts cannot readily understand their meaning [57]. Work by Jensen [69] encodes the reference scheme of a certificate policy using DocBook [138].

For policy retrieval, Blaze et al. created PolicyMaker, a tool that lets practitioners query policy actions using a database-like syntax [10]. Trcek et al. propose a DNS-like system to reference sets of security requirements [132].

No tools have been built to help with PKI policy creation. Klobucar et al., however, have stated the need for machine-assisted policy creation [72]. Furthermore, we note that during our collaborations, a mechanism to build policy was desired by both Motorola as well as by DigiCert.

Finally, the identity management research community has done some work in policy evaluation. Ball, Chadwick et al. have built systems to compute a trust index from XML-formatted CPS documents [5]. In addition, researchers at Trinity College, Dublin have thought about provisions of CP/CPS documents that are technically enforceable. O'Callaghan presented a suite of unit tests to measure the validity of a certificate relative to a policy [90].

2.1.3 Limitations of PKI Policy Analysis

At the start of this chapter, we introduced three core limitations of security policy management. We now reconsider each of these limitations in the context of the current approaches to PKI policy comparison.

Policy Gap Problem: There is a gap between traditional text-processing tools and the languages used in security policies.

In current practice, policy analysts operate on PKI policies by their reference

structure (defined in RFC 2527 and RFC 3647), but machine representations of policy such as PDF are organized by page. The page-based representation of policy imposes a semantic gap that forces policy operations to be largely manual because analysts must manually translate, in their heads, between policy page numbers and the reference structure.

PDFs may be built with bookmarks that are oriented to sections and subsections (the reference structure) of PKI policies, but PDF readers remain page-centric in their display of the text. Furthermore, the complexity of parsing this format not only gives PDF a large attack surface [144], but this complexity also prevents other services, such as braillers from easily processing text in PDF format [113].

The identity-management literature also exhibits symptoms of the gap between text-processing tools and the languages used in security policies.

There is a gap between policy formalization techniques in the literature and the ability to represent policy in a manner that supports policy decisions in real-world settings. Although the FPKIPA Technical Specification recommends writing CP and CPSs in a natural language, alternative representations of policies in the literature contradict this recommendation [46]. Data-centric XML, matrices, and ASN.1 require a person to read the source text and fit their interpretation of that text to a data format. Other literature agrees that the representations mentioned above are unsuitable for *relying parties*—practitioners and programs that use PKI certificates—to easily understand the meaning of a policy [18, 57].

Granularity of Reference Problem: Practitioners need to be able to process policies at multiple levels of abstraction and they currently cannot.

Although RFC 2527 and RFC 3647 define a hierarchical set of provisions, certificate-policy extensions for PKI reference only the entire document [63]. In order to address this need, the IGTF is creating *one statement certificate policies*: CPs that contain a

single provision.[1]

Operational PKIs must keep many layers of policy synchronized across standard policy formats, across multiple organizations, or between specification and implementation of policy. For example, analysts may want to update or compare a policy in 2527 format with a policy in 3647 format. In order to do this, policy analysts must be able to process CPs and CPSs in terms of individual provisions rather than entire passages. Similarly, the IGTF needs to be able to synchronize PKI policies across multiple *Policy Management Authorities (PMAs)* that include the *Asia Pacific Grid Policy Management Authority (APGridPMA)*, the *European Union Grid PMA (EU-GridPMA)*, and *The Americas Grid PMA (TAGPMA)*.

Policy Discovery Needs Problem: PKI security policy analysts and CAs need to be able to measure security policies to get feedback as to how individual organizations as well as higher-level bridges and grid federations change over time.

The timelines over which written policy and actual practice change are not in sync under current policy analysis practices. Policy evaluation in PKI compliance audit, grid accreditation, and bridge creation occurs infrequently and is inconsistent. Although compliance audits like WebTrust are supposed to occur every 6 months, in practice, audits usually happen much less frequently. In contrast, actual organizational practices can change more frequently than every 6 months.[2] When the IGTF changes an AP, members have 6 months to demonstrate that they are compliant with the new profile. Certificate Authorities that create and review policy want to know whether policy changes really do percolate through the federation in a timely manner and when they occur. Furthermore, diligent organizations who keep their policies current pose a challenge to bridge CAs who must manually map a member CP to the bridge baseline policy. Finally, when the policy of the bridge CA changes, the bridge

[1] In contrast, the IGTF could allow certificates to reference individual sections of a CP.
[2] Conversations with Scott Rea.

CA wants to be able to determine whether the current CPs of member organizations satisfy the new policy.

2.1.4 Section Summary

To Certificate Authorities and analysts in X.509 PKI, the term *security policy* refers to a natural-language legal document that specifies the creation, maintenance, and revocation of PKI certificates. PKI certificates are important and may be used for physical access to facilities or authentication to vast e-Science infrastructures. The level of assurance associated with a certificate depends upon the contents of a security policy.

Policy comparison is vital to determine the level of assurance of the certificates produced by a CA. PKI Compliance Audit, IGTF Accreditation, and Policy Mapping to Bridge PKIs all are X.509 processes by which analysts and CAs evaluate assurance relative to a set of baseline requirements. When a CA fails to perform any of these processes well, he exposes his organization to serious consequences that may include "business suicide".[3]

Despite the importance of these processes, there are drawbacks to the currently-practiced policy comparison process that are symptoms of the limitations of security policy analysis.

2.2 Network Configuration Management

Network administrators must write configuration files for network devices to implement new services such as *Voice Over IP (VOIP)*, or satisfy compliance or regulatory goals such as those defined by ISO 27001 [66] or NERC CIP [88]. If network administrators do not update configuration files, then organizations are either unable to

[3]Conversations with Scott Rea.

leverage new network capabilities or vulnerable to known attacks. If network administrators do update network device configurations, however, then they may introduce configuration errors. Many network outages, including Amazon's EC2 outage in 2011, are due to configuration errors [13, 92]

Our Fieldwork: During our research in network configuration management, we consulted with Paul Schmidt, a Senior Network Engineer at Dartmouth. Paul gave us access to five years of network device configuration files. We also consulted the expert opinion of Edmond Rogers, who secured electrical power control networks for a major *Investor Owned Utility (IOU)*.

2.2.1 Summarize and Measure Change

Network administrators and auditors need to be able to summarize and measure change in network configuration files to maintain and evaluate the security of a network.

Network administrators must be able to summarize and measure changes to a network because network administrators must routinely change the configuration of their network to accommodate new services and keep the network secure. The ability to *measure* changes is important because network auditors (for example in power-system control networks) may want logs to understand security posture. The ability to *summarize* changes is important because roles in router access-control policies may drift over time or be copied and renamed.

The consequences of not having tools to summarize and measure changes to a network are significant for administrators and auditors. Inadequate change and configuration management can increase risk of vulnerability [58] and decrease practitioners' ability to debug network misconfigurations. Misconfigurations can lead to network outages [92], exfiltration of data, or inappropriate access to network resources. If the

misconfigured network is a power control network, then network outages can lead to power outages.

2.2.2 Drawbacks of Current Approaches

Network administrators currently use tools such as the *Really Awesome New Cisco configuration Differ (RANCID)* to track which parts of a network configuration have changed [103]. Whenever a change is made to the running configuration of a router, RANCID automatically saves the old configuration in a version-control system and the network administrator provides a brief explanation of why the change was made. A line-based differencing tool generates an edit script and RANCID emails this script, along with a practitioner-provided explanation of the change, to all other administrators of that network.

RANCID is helpful, but the way in which it records changes hides the big picture. Practitioners cannot see how the network is changing nor can they quickly find changes that are relevant to a network administrator or an auditor. Although an administrator may receive a bug report that a network service started to behave strangely a few months ago, he cannot efficiently pinpoint what in the network changed. Instead, RANCID allows the practitioner to determine only which lines of configuration changed during those months.

In other words, administrators suffer because the change logs captured by RANCID are not easily turned into usable information to understand or measure network evolution.

Currently, utilities use change management products such as ChangeGear [22] and Remedy [106]. These products provide a ticketing system to document changes. These systems do not provide a mechanism to compare the contents of configuration files, nor do they allow one to measure how network primitives have changed. As such, these systems are only as good as the change documentation that the administrator

writes.

The quality of changes reported by a user may vary significantly. Writing change logs is time consuming, manual, and error-prone and the relevance of change logs may change over time. Although a network administrator might write change comments carefully, in three months those comments may no longer be helpful or relevant to another administrator or auditor. Practitioners often fail to write useful change documentation. Plonka et al. made this observation during their study of network configuration files and found that in a campus network, top commit comments included "initial revision", "asdf", and "test". They observed similar behavior in a service-provider network [98].

Recent findings from software engineering validate our concerns about the insufficiency of changelogs. In 2010, Israeli and Feitelson [67] looked at the evolution of the Linux kernel and argued for code-based measurements for software versus surveys and logs. They cite a study by Chen et al. that compares change logs for three software products and their corresponding changed source code; this study showed that 80% of the changes were not logged [24]. Another example comes from a 2007 study by Fluri et al. which looked at three open-source systems and how comments and code co-evolved. They found that newly-added code barely gets considered despite its considerable growth rate [43].

Splunk, a highly-scalable indexing engine, does allow practitioners to get a big-picture view of how data changes over time [121]. Splunk indexes changed configuration files so that administrators can search the data and construct search queries to identify events that reflect risky changes [122]. These search queries can then serve as alerts to the administrator. Splunk does not index information according to structures in a high-level language however, but instead tries to extract key/value pairs via regular expressions. One can then use these keys to refine searches within their organization's data.

Within the network configuration management literature, researchers have proposed several different ways to summarize and measure change. Two approaches that we will now discuss are longitudinal studies and metrics on router configuration files.

There are several papers in the literature that perform longitudinal studies on network configurations [25, 71, 98, 125, 126]. Sung et al. define blocks and superblocks to study correlated changes across router configurations [126]. Sun et al. look at the evolution of *Virtual Local Area Network (VLAN)* design to design algorithms that help practitioners design and refactor VLANs [125].

Plonka et al. studied the evolution of router configurations using the *stanza* for campus and server-provider networks over the period of 5 and 10 years respectively [98].

Kim et al. recently did a very complete longitudinal study of the evolution of router configurations, firewalls, and switches on two campuses over five years. They looked at the frequency of configuration updates, and identified correlated changes among other things [71].

Researchers also have proposed metrics on router configuration files. In their *Networked Systems Design and Implementation (NSDI)* 2009 paper, Benson et al. present complexity models and metrics to describe network complexity in a manner that abstracts away details of the underlying configuration language [8]. First, they present *referential complexity*, which captures dependencies between routing configuration components that may or may not span multiple devices. Second, they introduce a way to automatically identify roles within routers, and argue that the more roles a router supports, the more difficult a router is to configure. Finally, they present *inherent complexity* that quantifies the impact of the reachability and access-control policies on network complexity [8].

2.2.3 Limitations of Summarizing and Measuring Change

We now consider each of the core limitations of policy analysis in the context of summarizing and measuring change within network configurations.

Policy Gap Problem: There is a gap between high-level language constructs used by network administrators and the low-level lines upon which their tools operate.

For example, consider *Cisco's Internetwork Operating System (Cisco IOS)*'s `include` command.[4] The `include` command lets practitioners sift through router files in terms of lines even though the Cisco IOS language is hierarchical by design. Alternatively, the RANCID tool only reports comparisons in terms of lines, a low-level language construct.

Line numbers are not the best choice to measure change between multiple versions of network configurations. Practitioners need tools to quantify change in terms of the hierarchical structure of network device configuration languages (such as Cisco IOS) in order to understand trends in how a network changed. Line numbers are highly volatile between multiple versions of a file.

RANCID may report meaningless changes that add noise to changelogs. For example, if one permutes five lines in a block of a configuration file, then RANCID will report it as 5 deletions and 5 insertions regardless of whether the behavior of the configuration is unchanged.

Granularity of Reference Problem: Network administrators and auditors need to be able to process policies at multiple levels of abstraction and they currently cannot.

Traditional Unix `diff`, upon which RANCID is based, may contribute to unnecessarily long edit scripts when it expresses a change at the level of the line. Consider

[4] Conversations with Enno Rey. Enno Rey is a managing director and founder of ERNW, a German security company that runs the TROOPERS security conference [131].

an example from Cisco IOS in which a revision could be expressed as deleting and then inserting lines or inserting and deleting a single interface block. The latter is more meaningful and also reduces the amount of change information through which a practitioner must wade.

Policy Discovery Needs Problem: Network administrators and auditors need feedback on how network security policies and related security artifacts change over time. Administrators also need to identify specific changes that cause *bad* behavior. Lim et al. noted that "top-down security policy models are too rigid to cope with changes in dynamic operational environments" [79]. Furthermore Sun et al. note that the frequency and complexity of configuration changes (due to the addition of new hosts, movement, reorganization of departments and personnel, revision of security policies, and upgrading routing hardware) makes the overall process of redesigning and reconfiguring enterprise networks error-prone [125].

In a top-down approach to network security policy, an organization's security expert (perhaps a *Chief Information Security Officer (CISO)*) specifies an organization's security-policy controls and network administrators implement those controls within a configuration language. Practitioners need the capability to measure a network's *actual* security posture from a set of network configurations. The NetAPT [89] tool provides a bottom-up approach to policy. Practitioners use NetAPT to formally verify the reachability constraints of a network topology inferred from firewall configuration files. Our research is complementary because it allows practitioners to measure network security properties other than reachability. Bottom-up approaches to policy close the feedback loop so that practitioners can measure security posture. This feedback will enable practitioners to cope with dynamic environments and continually understand how their networks change.

2.2.4 Section Summary

To network administrators and network auditors, the term *security policy* refers to a network device configuration that specifies the security posture of a network. Network administrators must routinely update network configurations to accommodate new services and keep the network secure and compliant.

The security policy analysis task of change summarization and measurement is essential in order for administrators to maintain and for auditors to evaluate the security of a network. Insufficient change control and configuration management increases the risk of misconfigurations and makes networks harder to debug [58]. Misconfigurations lead to network outages [92]. The Amazon EC2 outage of July 2011 [13] cost $5600 per minute [62]. When the network is a power-control network or a hospital network, a network outage may even result in civil unrest or the loss of life.

Despite the importance of change summarization and measurement, there are drawbacks to the currently-practiced comparison process such as insufficient change logs. These drawbacks are symptoms of the three core limitations of policy analysis introduced in Chapter 1.

2.3 Electrical Power Grid

In the domain of the electrical power grid, the term *security policy* may refer to North American Electric Reliability Corporation's Critical Infrastructure Protection (NERC CIP) regulatory guidelines or to a security primitive in a configuration language for an Intelligent Electronic Device (IED) [88].

The smart grid will increase the stability and reliability of the grid overall with vast numbers of cyber components but these cyber components also have the potential to increase the attack surface of the grid.

The smart grid is already large and complex and as we add more devices to the internet, utilities and auditors must be aware of security. The power grid has been called the "world's biggest machine" by engineers and is part of the "greatest engineering achievement of the 20th century" [87]. Even a high-level view of the smart grid shows a great deal of complexity [58].

If we look at just the customer domain in 2010 there were approximately 160 million residences in the US [102] and only 18 million smart meters deployed in 2012 [4]. This discrepancy between the number of residences and the number of smart meters indicates that there are many more smart meters that will be deployed in the future. Furthermore, there were 250 million registered cars in 2010 [85] and as more cars become electric, we will see an increasing number of devices on the smart grid. For example, car batteries may be used to store electricity and the power grid will need to be smart in order to coordinate vehicle charging with other electrical loads.

Similarly, if we consider just one Investor Owned Utility (IOU) studied in 2009 at Idaho National Laboratories [6], we can see that one IOU was in charge of 200 substations and 1,000,000 residential customers. During the 2012 Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) Industry Day, I spoke with an individual that worked with an IOU of 700 substations and 5,000,000 residential customers.

In the future, the number of devices on the smart grid will increase and each of these devices will produce an *avalanche* of disparate data because these devices need to be configured, send and recieve data via a variety of protocols, and log information.

Our Fieldwork: Throughout our research in the power grid, we routinely consulted with Edmond Rogers. Edmond Rogers used to secure the power-control network of a major *Investor Owned Utility (IOU).* He currently serves as the utility expert on the DOE-funded TCIPG project. In addition, we actively sought feedback from our work

from academia and industry when we demonstrated our work at TCIPG Industry Day, and presented a poster at the DOE Smart Grid Cybersecurity Information Exchange.

2.3.1 Change Control and Baseline Configuration Development

For utility administrators and auditors, change control and baseline configuration development are important processes that have serious consequences if these processes are done poorly.

Power system control networks must comply with NERC CIP regulations and those regulations require utilities to meet change control and configuration management requirements. Specifically NERC CIP-005-4a R5.2 requires responsible entities to update documentation within 30 days of a change to the network [88]. This ensures that documentation reflects the actual configuration of the power system network. CIP 003-4 R6 requires utilities to "establish and document a process of change control and configuration management to add, modify, replace, or remove Critical Cyber Asset hardware or software, and document all entity or vendor-related changes to hardware and software components of Critical Cyber Assets pursuant to the change control process" [88].

The NERC CIP regulations also require utilities to develop baseline configurations. Specifically NERC CIP 010-1 R1.1 requires utilities to develop a baseline configuration of devices on their control networks that includes information such as "physical location, OS(s) and versions, any commercially available application software (and version), any custom software and scripts installed, logical network accessible ports, and security-patch levels" [88].

The consequences of insufficient change control and baseline configuration development are significant, especially as the number of devices on the smart grid increases. In general, the consequences of failing to fulfill NERC CIP regulations are severe and

scale up to $1.5 million for every day of the violation. In addition to financial consequences, failure to comply implies a lack of basic and sound security controls. A lack of sound security controls makes the control network vulnerable to cyber attacks and their consequences, which include power outages.[5]

The consequences of insufficient change control and baseline configuration development will increase with the number of devices on the smart grid. According to the NISTIR Guidelines for Smart-Grid Cyber Security, "increasing the complexity of the grid could introduce vulnerabilities and increase exposure to potential attackers and unintentional errors" [58]. This observation is consistent with the network configuration management; Benson et al. note that complexity of a network increases with maintenance and changes to configurations [8].

2.3.2 Drawbacks of Current Approaches

As mentioned in our previous section, utilities use tools such as RANCID to monitor changes to Cisco IOS devices. These devices may include switches, firewalls, and routers. Utilities also use tools such as TripWire [91] to monitor changes to a variety of general-purpose computers on the network. For example, Remote Terminal Units (RTUs) allow practitioners to visualize information that comes from various IEDs are general-purpose computers. The commercial TripWire product monitors changes to a set of file attributes and couples this with a line-based `diff` tool. TripWire stores hashes of software to be monitored and reports when the stored hash and periodically-recomputed hash differ. This technology, however, only informs utilities *whether* a change occurred, not *how* the software changed.

Utilities use tools such as ChangeGear [22] and Remedy [106] to record changes to devices on a control systems network. These products do not provide a mechanism to compare the contents of files, nor do they automatically document how the file

[5]Conversations with Edmond Rogers.

was changed. Instead, these change ticketing systems rely upon humans to create changelogs. As we saw in the last section, changelogs are insufficient because they are error prone and time consuming to write.

There are not many tools available to help practitioners to develop and compare baseline configurations. During conversations with a utility expert, we learned that many utilities use spreadsheets to manually document baseline configurations of systems (such as version number and software installed).[6]

2.3.3 Limitations of Change Control and Baseline-Configuration Development

We now demonstrate how the drawbacks of change control and baseline-configuration developement are symptoms of our three core limitations of security policy (and security artifact) analysis. In the process, we will also align each of these core limitations with the *Roadmap to Achieve Energy Delivery Systems Cybersecurity*, a strategic framework for smart grid cybersecurity produced by the Obama Administration [7]. For the remainder of this thesis, we will refer to this strategic framework as *Roadmap*.

Policy Gap Problem: The current state of the practice in change control and configuration management suffers from a gap between the languages in which security policy artifacts are represented and the tools available to process those artifacts.

In the previous subsection, we discussed how current approaches to change control allow utilities to monitor *whether* control network devices have changed but not *how* those devices changed. Furthermore, change documentation relies upon ticketing systems that require users to manually report changes and this is a slow, manual, and error-prone process. Additionally, utilities employing state of the practice techniques in baseline configuration currently use spreadsheets to document systems.

[6] Conversations with Edmond Rogers.

Unfortunately, these manual approaches will not scale. In the future, the number of devices on the smart grid will increase and each of these devices will entail an avalanche of disparate data because these devices need to be configured, send and receive data, and log information.

In the context of the power grid, data formats and protocols such as the *Common Information Model (CIM)* [33], Cisco IOS, *Energy Systems Provider Interface XML (ESPI-XML)* [41], GOOSE messages, *Substation Configuration Language (SCL)* [64], and Windows Registries all encode hierarchical object models.

In order to understand *how* devices on a control network have changed and in order to baseline security policy in a scalable manner, practitioners need tools that can process the variety of disparate data in terms of hierarchical object models. Practitioners currently lack such tools. In fact, the 2011 *Roadmap* states that a barrier to assess and monitor risk in the smart grid is the inability to provide "actionable information about security posture from vast quantities of disparate data from a variety of sources and levels of granularity" [7].

Granularity of Reference Problem: The previous quote from the *Roadmap* also recognizes the need to be able to process data at multiple levels of granularity. The drawbacks of change control and baseline configuration development discussed above are symptoms of the need to be able to process policy artifacts at multiple levels of abstraction.

Practitioners need to be able to process policy artifacts at multiple levels of abstraction because policy artifacts are encoded in hierarchical object models. For example, Cisco IOS is a hierarchically structured network-device configuration language that contains primitives for *Access-Control Lists (ACLs)*, roles—logical groupings of users, devices, or protocols (in Cisco IOS roles are encoded as object groups). In addition, the *International Electrotechnical Commission (IEC)* 61850 standard de-

fines Substation Configuration Language (SCL) which can be used to define an IED's state and capabilities such as access-control settings and available communications interfaces. [64].

Current change-detection control systems such as TripWire [91] are only able to determine *whether* and not *how* a file has changed because they operate on file system objects. In order to determine *how* and *where* a file has changed, practitioner tools need to be able to process policy artifacts at multiple levels of abstraction. Using this capability, an administrator could set up a rule to alert users when an ACL on a network device changed and to ignore changes to network interfaces.

The ability to process policy artifacts at multiple levels of abstraction can also address the shortcomings of change control systems. Practitioners would not have to rely on manually-generated changelogs and reports if they could rely on tools that automatically detected and summarized changes to devices at an appropriate level of abstraction. In addition, this capability does not require utilities to store changelogs, which can be prone to log injection and record falsification [58]. Furthermore, changelog storage space can be reduced by reporting changes at different levels of abstraction; for example, adding a new role in an IED's access-control policy can be reported as the addition of single security primitive (such as a role) rather than as the addition of 10 lines.

The current approaches to create and compare baseline configurations are inefficient and will not scale even though this capability is required by NERC CIP 010-1. If practitioners use spreadsheets to record changes to devices on a control network, practitioners must create logs manually. We have already mentioned that manually-created changelogs are insufficient in the domains of network configuration management and software engineering. In addition, a majority of power control systems use Windows and NERC CIP regulations were written for Windows-based control systems. Windows Registries encode most of the pieces of information required for

a baseline configuration in a hierarchically-structured set of key/value pairs. Currently, practitioners cannot operate on individual components of these registries even though they would like to. At the 2012 TCIPG Industry Day, a scientist at Honeywell lamented that he had to use a line-based diff to compare hierarchically-structured Windows Registries.[7]

Policy Discovery Needs Problem: The NERC CIP requirements for change control and configuration management and for baseline-configuration creation and comparison point to the need for feedback on how the security posture of power control networks changes. Utilities and auditors need the ability to measure big-picture trends in power system control networks. Utilities and auditors alike also need the capability to pinpoint specific changes that are outliers within those trends. As described by one auditor and penetration tester, utilities and auditors need the ability to measure *normal*—current methods do not allow them to know what *normal* is.[8]

Currently practiced techniques to detect and document changes rely on processes that are too coarse to allow practitioners to measure change at the desired level of abstraction and too manual to perform audits on a more frequent basis [58].

The drawbacks of current business processes to satisfy these NERC CIP requirements can be addressed by a feedback loop for security policy that relies on the ability to measure how configuration files change at multiple levels of granularity. This vision is consistent with the *2012 Smart Grid Program Overview* by the director of the Smart Grid and Cyber-Physical Systems Program office who called for "new measurement methods and models to sense, control, and optimize the grid's new operational paradigm" [4].

[7]Conversation with S. Rajagopalan. S. Rajagopalan is a scientist at Honeywell.
[8]Conversation with Bryan Fite at TCIPG Industry Day, 2012.

2.3.4 Section Summary

To utility administrators and auditors, the term *security policy* may refer to NERC CIP regulatory guidelines or a security primitive in the configuration language of a device on the power grid.

The NERC CIP provisions for change control and configuration management and baseline configuration creation and comparison are important operations on security policy artifacts that can have significant and even severe consequences if done improperly.

Despite the importance of these processes, the current practices to satisfy these NERC CIP provisions rely on processes that measure changes to policy artifacts at either a very low level (the line) or a very high level (the file or network). Furthermore, many of the change management approaches rely upon manual documentation, which is error-prone and will only become less reliable as the number of devices on the smart grid increases.

2.4 Conclusions

The term *security policy* can mean many different things to practitioners in a variety of domains that include identity management, network configuration management, and the electrical power grid. Failure to manage security policies has severe consequences that can include inappropriate access to sensitive information, network outages. We have seen that regardless of the domain, security policies may be viewed as a structured text. Many of the drawbacks of current practices to analyze policies or associated security artifacts are symptoms of the Policy Gap Problem, Granularity of Reference Problem, and Policy Discovery Needs Problem. The next chapter introduces several concepts from language theory, parsing, and discrete mathematics to address these limitations and formalize the notion of *structured text*.

Chapter 3

Theory

We can directly address our three core limitations of security policy analysis with concepts from language theory, parsing, and discrete mathematics. In this chapter, we provide background information to explain these concepts and relate them to our core limitations.

The first section of this chapter is a quick introduction to basic concepts from language theory, parsing, and discrete mathematics so that the thesis is self-contained. More experienced readers may skip ahead to the second section. For a full treatment of the topics, readers should consult a textbook such as Sipser [115] for language theory, a book on compilers for parsing [2], and a book on discrete mathematics for a discussion of sets and set operations [108].

In the second section of this chapter, we will demonstrate how we can apply these concepts from computer science and mathematics to formalize text processing and thereby address our three core limitations of security policy analysis (1) the gap between tools and policy languages, (2) the inability to process policy at multiple levels of abstraction, and (3) the need for a feedback loop to measure security policy evolution.

3.1 Theoretical Background

In order to formalize security policy analysis, we first must understand the languages in which security policies and associated policy artifacts are written. Therefore, we begin with a definition of *language*.

3.1.1 Language Theory

Language theory builds on the notion of a *set*: an unordered collection of objects or *elements*. The elements contained within a set are unique. A set is *nonempty* if it contains one or more elements. If there are exactly n distinct elements in the set S where n is a nonnegative integer, then S is a *finite set* and n is the cardinality of S. Language theory defines an *alphabet* as any nonempty finite set. The elements of an alphabet are called *symbols*.

Language theory also builds on the notion of a *sequence*: an ordered collection of elements in a set. In contrast to a set, elements in a sequence do not have to be unique. In language-theory, a *string* is a sequence of symbols over an alphabet. The *length* of a string w, written as $|w|$, is the number of symbols it contains. A string of length zero is called the *empty string*, and is written as ϵ [115]. Figure 3.1 illustrates how "Doubleday" satisfies the properties of a string over the English alphabet. The length of the string "Doubleday" is 9 symbols.

A *language* is a set whose elements are strings over some fixed alphabet. For example, both the empty set $\emptyset$ and the set containing the empty string ϵ are languages. Consider the collection of *last names of people in the Baseball Hall of Fame*. Figure 3.2 illustrates this language.[1] We note that two players with the same last name are represented by a single element in this language since language elements are unique. In this language the string "MacPhail" represents both Larry and Lee MacPhail, the only father-son pairing in the Hall of Fame [101].

[1] Unless otherwise noted, the sets in our figures do not depict exact cardinalities.

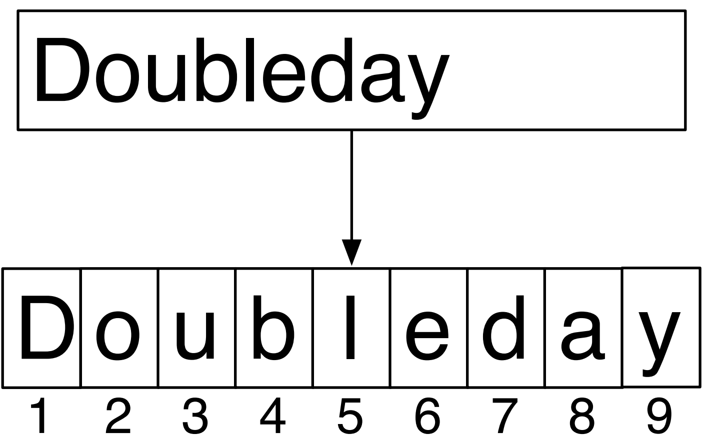


Figure 3.1: A *string* is a sequence of symbols taken from some alphabet. The string *Doubleday* in this example is a sequence of symbols from the English alphabet ordered from left to right (reading order).

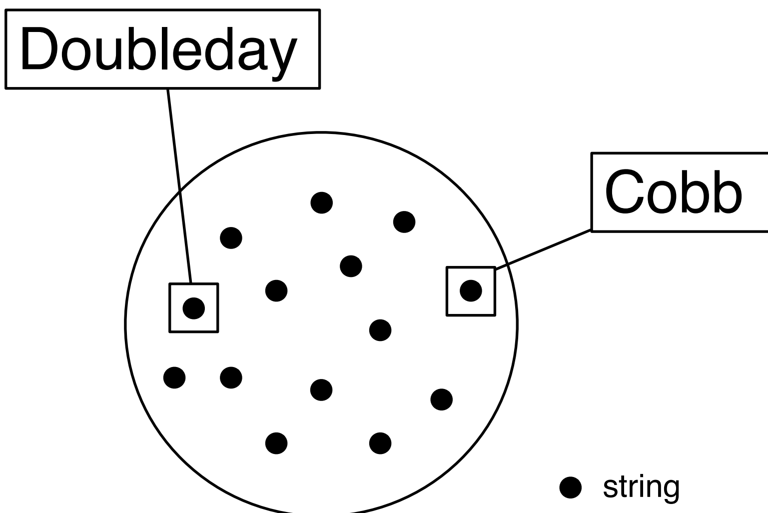


Figure 3.2: A *language* is a set of strings. This figure illustrates the language of last names of people in the Baseball Hall of Fame.

Languages can be more complex than sets of single words. We can use languages to group strings with common structure. For example, the language of properly-nested parentheses contains the elements "([])", and "{[]([])}[{}]", but not "(]". Also, consider the language of sentences that consist solely of a one-word noun, followed by the verb *are*, followed by a one-word adjective and a period. Strings in this second language include "Bears are fierce." and "Trampolines are fun." but not "The Bears are fierce." and not "Trampolines are fun to eat.". Even more interesting, we can use languages to represent non-textual structures such as DNA.

A prerequisite of textual analysis is the ability to recognize whether a string (at some level of abstraction) belongs to a language or not. For example, a network administrator may be interested in the set of roles defined by a firewall's access-control policy. The set of strings that represent roles is a language in the language-theoretic sense.

However, textual analysis also requires practitioners to understand the properties of strings within languages and how they relate to other languages. For example, we may be interested in all boys' names that begin with a letter in the first half of the alphabet and who are currently first-year undergraduates at Dartmouth. The former criterion is a property of strings in the set of boys names and the latter criterion requires us to relate the language of boys' names to the language of class years in the Dartmouth College roster. We can formalize this aspect of textual analysis with the notion of a *datatype* or *type* which combines a set with operations that can be performed on elements contained by that set. For textual analysis, we can define a set of operations appropriate for a language.

In computer science, we construct computational machines, called *recognizers*, that accept or reject a string as being an element of a language. Figure 3.3 illustrates a recognizer. Since a recognizer is a computational machine, there is a nice connection between the complexity of the language and the complexity of the machine. In

fact, language theory categorizes languages into different classes depending upon the type of computational engine required to recognize the language. Figure 3.4 relates language complexity to machine complexity for two language classes: regular and context-free languages.

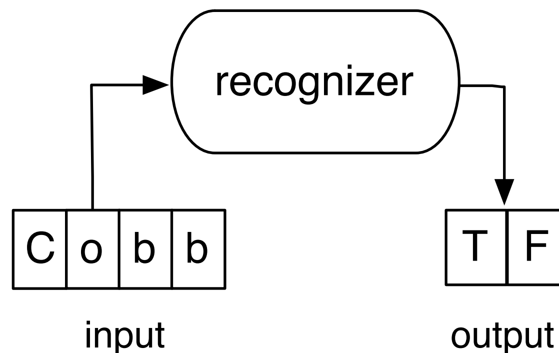


Figure 3.3: A *recognizer for a language* is a computational machine that outputs TRUE or FALSE if an input string is in the language or not. The string *Cobb* is in the language of last names in the Baseball Hall of Fame.

Regular Languages

The set of credit-card numbers, phone numbers, IP addresses, email addreses are all regular languages. A language is *regular* if it can be recognized using a *regular expression*. Regular expressions are a notation through which recognizers of languages can be defined. The formal definition of *regular expression* over some alphabet Σ consists of the following three rules:

- ϵ is a regular expression whose language is $\{\epsilon\}$, the set containing the empty string.
- If $a \in \Sigma$, then a is a regular expression whose language is $\{a\}$, the set containing the string a.

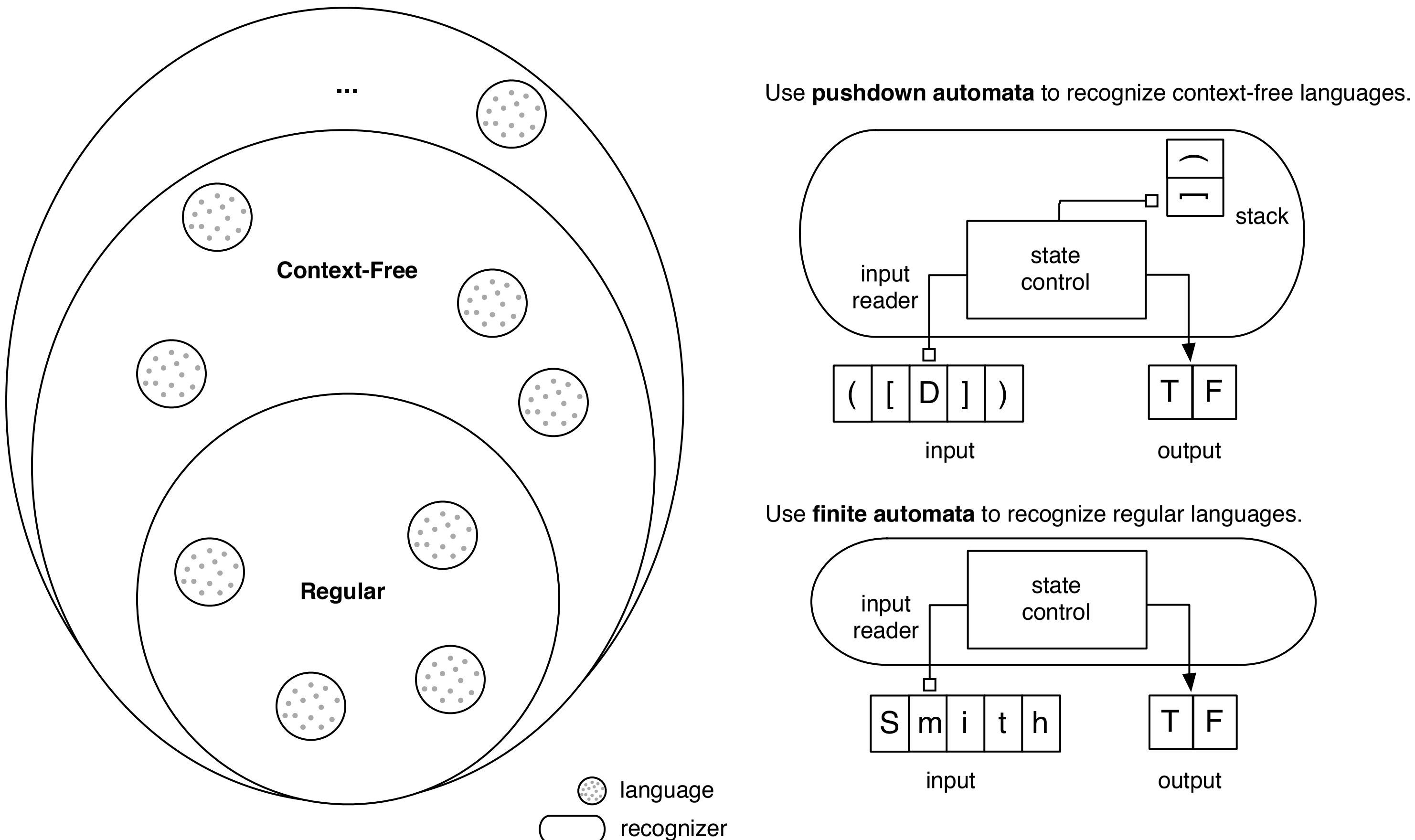


Figure 3.4: Language theory categorizes languages into different classes depending upon the complexity of the recognizer. In this diagram we see schematics for a *finite automaton* and a *pushdown automaton* that recognize regular and context-free languages respectively. A *pushdown automaton* is a finite automaton with a *stack*. Therefore, a pushdown automaton can do whatever a finite automaton can do, and more (although it's not always true that adding features to a machine increases its recognition power). Readers who want to learn more about the details of these machines should consult a language-theory textbook. We base our diagrams of these machines on the presentation in Sipser's book [115].

- If r and s are regular expressions whose languages are $L(r)$ and $L(s)$ then we can define the following languages:

  1. $(r)|(s)$ is a regular expression whose language is $L(r) \cup L(s)$
  2. $(r)(s)$ is a regular expression whose language is $L(r)L(s)$
  3. $(r)*$ is a regular expression whose language is $(L(r))*$
  4. (r) is a regular expression whose language is $L(r)$

We can convert a regular expression into a computational engine called a *finite automaton*. If a language cannot be described by a regular expression, then it is not a regular language.

Benefits and Limitations: Regular expressions are relatively simple to write and as such, are useful for practitioners that want to extract a regular language from some input text. In fact, the utility of `grep` and `sed` one-liners stems from the ability for practitioners to construct finite automata quickly and easily with the notation of regular expressions. For example, the regular expression *live free* | *die* denotes the language that contains only two strings: "live free" and "die".

Unfortunately, regular expressions do not recognize all the kinds of languages that practitioners might want to process in the real world. One limitation of regular expressions is that they cannot solve the *parentheses-matching problem*: recognize the language of strings with properly-nested parentheses. The parentheses-matching problem shows up frequently in the real-world. For example, both natural-language documents as well as programming languages have hierarchical data models that can contain recursively-nested blocks of text.

Context-Free Languages

The set of *context-free languages*, in contrast to regular languages, include languages that possess a recursive or hierarchical structure. We call these languages context

free because their elements are generated by substituting strings for variables called *nonterminals* regardless of the context in which those nonterminals occur. A language is *context free* if it can be generated using a *context-free grammar*. *Context-free grammars* are a mathematical formalism to specify recursive or hierarchical structure.

More formally, a context-free grammar consists of terminals, nonterminals, a start symbol, and productions.

- *Terminals* are the basic symbols from which strings are formed. The three terminals of grammar G in Figure 3.5 are c, o, and b respectively.

- *Nonterminals* are syntactic variables that denote sets of strings; one of these nonterminals is called the start symbol. As mentioned above, we call these languages *context-free* because we can substitute strings for nonterminals regardless of the context in which the nonterminal appears.[2] In Figure 3.5, the grammar G consists of three nonterminals S, O, and B.

- A grammar's *start symbol* is a unique nonterminal whose language is the language defined by the grammar. In Figure 3.5, the S nonterminal is the start symbol.

- *Productions* of a grammar specify rewrite rules to transform a nonterminal into a string of nonterminals and terminals. If the grammar is in Chomsky Normal Form (CNF), the resultant string may not contain the start symbol and the rule $S \to \epsilon$ is in the grammar. Each of the nonterminals in Figure 3.5 has a production and the rule $S \to \epsilon$ is implicit in the grammar for this example.

Pushdown automata are computational engines that recognize input strings with arbitrarily-deep hierarchical structure. A *pushdown automaton* extends a finite automaton by adding a *stack*. The *stack* adds additional memory to the machine with

[2] This property motivates *context-sensitive languages* in which nonterminal substitution depends on its context. Context-sensitive languages require an even more powerful recognizer.

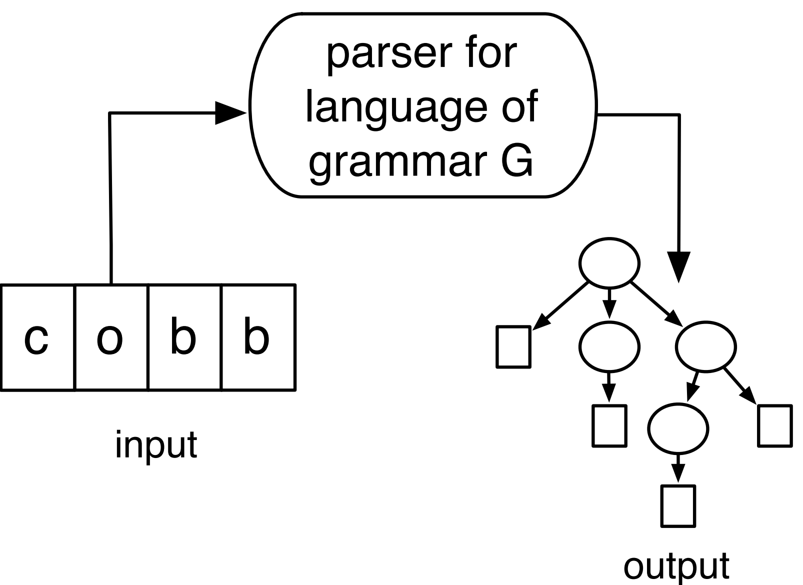


Figure 3.5: A *context-free grammar* consists of a set of terminals (c, o, b), nonterminals (S, O, B), and productions that specify how to rewrite nonterminals as strings. A *parser* recognizes the language of a grammar by trying to produce a parse tree for an input string. The string is in the language of the grammar if the parser can produce a parse tree for the string. The parse tree in this figure is shown in more detail in Figure 3.6.

the specific purpose of keeping track of where the current input symbol is with respect to the hierarchical structure of the input already processed. In Figure 3.4, we see that the pushdown automaton is currently processing the D symbol and has already seen the first two open parenthesis. The stack reader therefore points to the level of the hierarchy in which D resides.

Benefits and Limitations It is no coincidence that practitioners often need to recognize recursive or hierarchical structures in configuration and programming languages because the syntax of many programming languages was traditionally specified via context-free grammars. In addition, as languages evolve and acquire new constructs, grammatical descriptions of languages can be extended easily. Finally, all regular languages are also context free and so we can express regular languages via a grammar as well.

Although context-free grammars are able to specify a proper superset of languages specified by regular expressions, they still suffer from several limitations. First, context-free grammars traditionally are harder to write and so are more commonly seen in compiler construction than in text-processing toolchains despite the increase in high-level languages.

A second limitation of context-free grammars is that they do not recognize all the languages that we may encounter in the real world or even in real-world security policies. As mentioned in a previous footnote, *context-sensitive* languages are a proper subset of context-free languages and require an even more powerful recognizer. For example, the full Cisco IOS language is context sensitive [16]. Despite this, a meaningful subset of a context-sensitive language may be recognized via regular and context-free grammars. Another example of a language that cannot be recognized by a context-free grammar is an abstraction of the problem of checking that the number of formal parameters in a function definition matches the number of parameters to a

call to that function [2].

3.1.2 Parsing

In compiler construction, a *parser* is a software module that recognizes whether a given string is in the language of that grammar and determines its grammatical derivation. In fact, some (but not all) parsing models closely mimic the architecture of a pushdown automaton and explicitly maintain a stack of grammar symbols applied. Given an input string and a grammar that specifies a language, a parser attempts to construct a *parse tree*. If the parse tree can be constructed, then the input string is in the language of the grammar. Otherwise, the input string is not in the language of the grammar. Figure 3.5 illustrates a parser that recognizes the language of grammar G.

A *tree* T consists of a set of *nodes* and each vertex may have zero or more children. A *parse tree* is a *rooted, ordered tree* that encodes the application of the productions in a grammar that are required to produce or *derive* a string in the language of that grammar from the grammar's start symbol. Figure 3.6 diagrams a parse tree for the input *cobb* parsed with respect to grammar G.

If a node v is a child of another node p, then we call p the *parent* of v. There are 8 nodes in the tree of Figure 3.6. In Figure 3.6, node 7 is a child of node 8 and node 7 is the parent of nodes 5 and 6. Nodes that share the same parent are called *siblings*. Nodes that have no children are called the *leaves* of a tree. In our running parse-tree example, the leaves of the tree correspond to terminals of the grammar G.

Often, a particular node of a tree may be designated as the *root*, the node which has no parent. When T is a parse tree, the root is labeled with the start symbol of the grammar.

If v is a node in a tree, we can define a *subtree* T_v by taking v as the root.[3] In

[3] We will later overload this notation to refer to parse tree T_w of a string w when parsed with

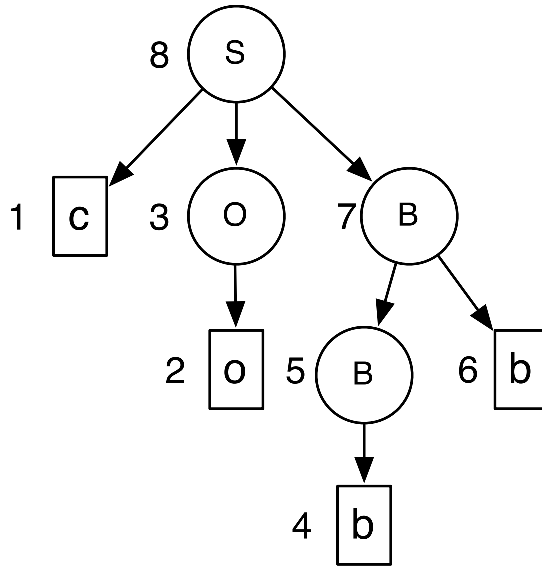


Figure 3.6: A *parse tree* is a *rooted, ordered tree* that encodes the application of productions in a grammar required to produce an input string. This diagram illustrates the parse tree for the input *cobb*. Informally, G specifies the language of strings made of one c followed by one or more o's and then one or more b's.

practice, the leaves of a *parse subtree* correspond to substrings in the language of the subtree root v's production. Furthermore, we can view these substrings as belonging to a language. This language has a grammar whose start production is $S' \rightarrow X$. S' is the new start symbol and X is the nonterminal with which v is labeled. For example, node 3 in Figure 3.6 is the root of the of substring of all o's in the input string while node 7 is the root of the substring of all b's in the input string.

If the children of each node are ordered, then we call T an *ordered tree*. Since each non-leaf node (*interior node*) in the parse tree is labeled by some nonterminal, the children of the node are ordered, from left to right, by symbols in the right side of the production by which A was replaced to derive the parse tree [2]. For example, Figure 3.6 node 7 is labeled with production B and the children of node 7 are ordered according to the production $B \rightarrow Bb$.

A grammar is *ambiguous* when it generates more than one parse tree for a given

respect to a grammar.

string. For some parsing purposes, ambiguity is undesirable and so different rules allow one to narrow down the set of possible parses to a single parse tree.

3.1.3 Security Policy Corpora as Data Types

In Chapter 2 we observed that security policies and related policy artifacts are expressed in a variety of formats ranging from natural-language legal documents written according to RFC 2527 or 3647, to configuration files written in Cisco IOS, to IED capabilities encoded in the IEC 61850 Substation Configuration Language (SCL). Each of these sets of files forms a *corpus*, a collection of texts that policy analysts want to analyze.

We formally represent a corpus as a *datatype*, a language paired with a set of operations whose operands are strings in that language. For the purpose of our research, the set of RFC 3647 policies, the set of configuration files written in Cisco IOS, or the set of IED capabilities written in SCL are three languages upon which we may define different string operations.

In our research, these string operations may implement traditional set operations such as element equality or union. Element equality is interesting because different implementations of equality allow us to *partition* a language into *equivalence classes*. A *partition* of a set S is a collection of disjoint, nonempty subsets of a set S so that they have S as their union. Figure 3.7 illustrates two ways that we could define equality.

We may define string operations that allow us to either extract other datatypes according to a data format's hierarchical object model or to encode a practitioner's analysis technique. Figure 3.8 illustrates how we may use a *parse* operation to extract the set of roles contained within a set of Cisco IOS network configurations. Each element in the input set is the contents of a network configuration file (a string). For each string element in the input set, we scan the string to extract all occurrences of

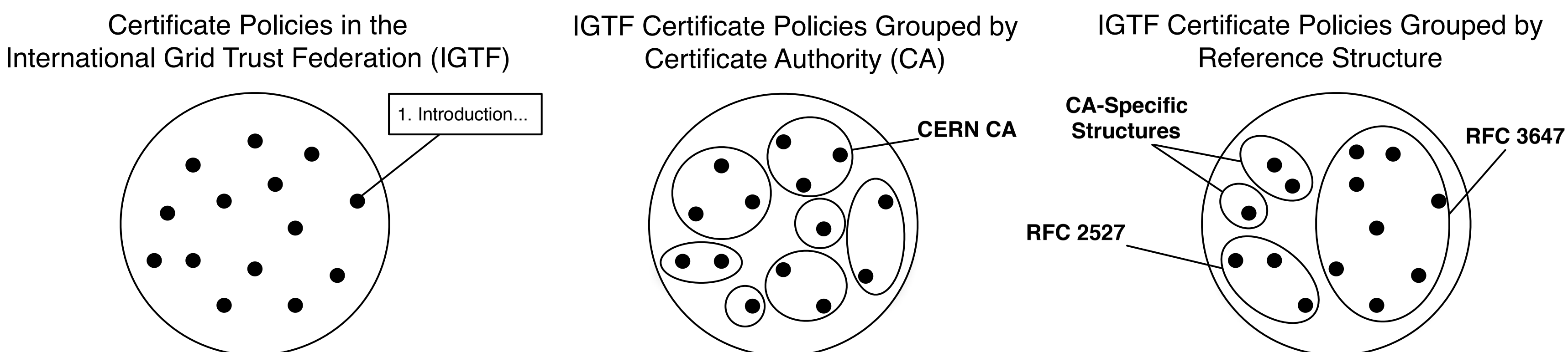


Figure 3.7: By defining the *equality* operation on strings in a language in different ways, we can *partition* a language into different *equivalence classes*. For example, we could define two strings over the English alphabet to be *equal* if they were identical after we converted the characters to lowercase and stripped stopwords such as *the* and *and*. In this figure, we define the equality of PKI policies from the International Grid Trust Federation (IGTF) in two different ways. First, two policies are equal if they were written by the same Certificate Authority. In the second example, two policies are equal if they both have identical sequences of section headers. Standard section headers are defined in RFC 2527 and RFC 3647.

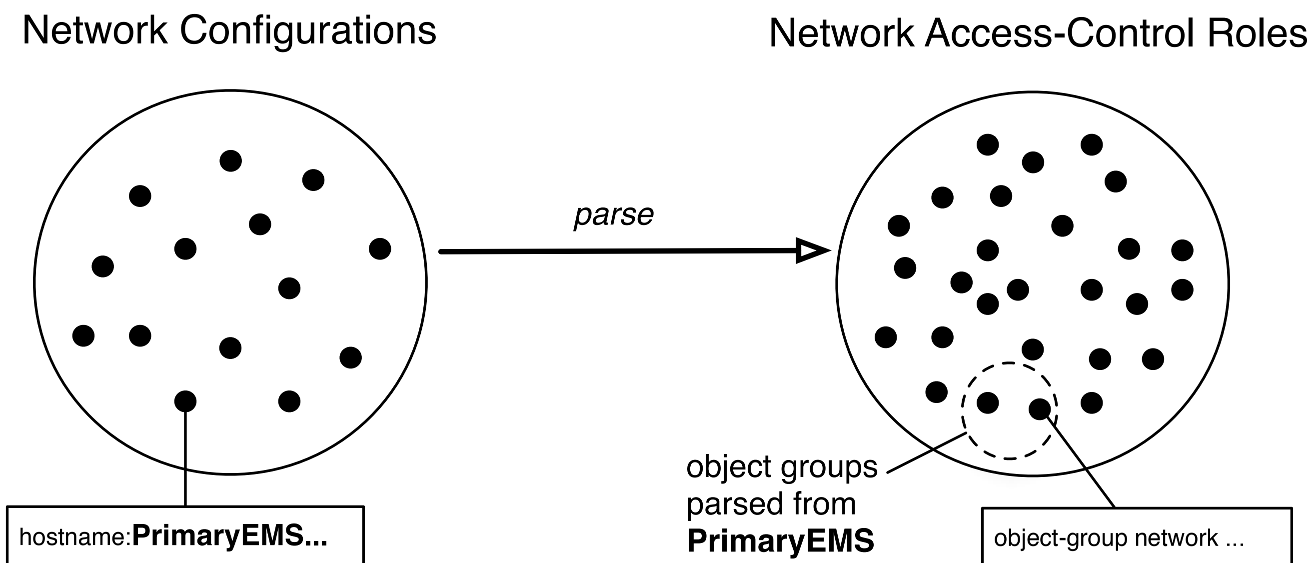


Figure 3.8: We can use the *parse* operation to extract and analyze high-level language constructs. Given a set of network device configurations, we can parse every string element in the set to extract all occurrences of an object group. Object groups are similar to roles in an access-control policy. This figure shows that the Primary EMS firewall configuration defines two roles.

a role definition. Alternatively, given the same set of network configurations, we may choose to issue the *parse* command to extract the set of roles that contain a particular device or protocol, and thus encode a practitioner's role analysis technique.

Finally, we may define string operations in a manner that allows us to compare strings in a language via simple counting or actual distance metrics.

In order to compare strings in a language via simple counting we could define a function whose domain is the strings in the language and whose domain is the number of lines contained in each string. This function would allow us to count the number of lines within a security primitive such as a role. This simple approach would help auditors pinpoint complicated configurations and administrators identify where it might be necessary to refactor the policy.

Alternatively, we could use string and tree distance metrics to measure the distance between strings in a language and parse trees in context-free languages. One distance metric that we use for string and trees is *edit distance*.

The *edit distance between two strings* s_1 and s_2 is the minimum number of *edit operations* necessary to transform s_1 into s_2. Traditional string edit distance algorithms use delete, insert, and substitute as their edit operations. A sequence of edit operations used to transform s_1 into s_2 is called the *edit script*. For example, the string edit distance between the strings *cobb* and *cob* is 1 character because by deleting a single character *b*, we transform the former into the latter.

The *edit distance between two trees* is the minimum number of *edit operations* necessary to transform one tree to another. The edit operations that we consider consist of deleting, changing, and appending tree nodes. Again, a sequence of edit operations between the two trees is called an *edit script*. Figure 3.9 illustrates that we can apply tree edit distance to the parse trees of strings in a context-free language. In the example depicted, the tree edit distance between the parse trees for *cobb* and *cob*, T_{cobb} and T_{cob} respectively, is 2 tree nodes. The distance is 2 rather than 1

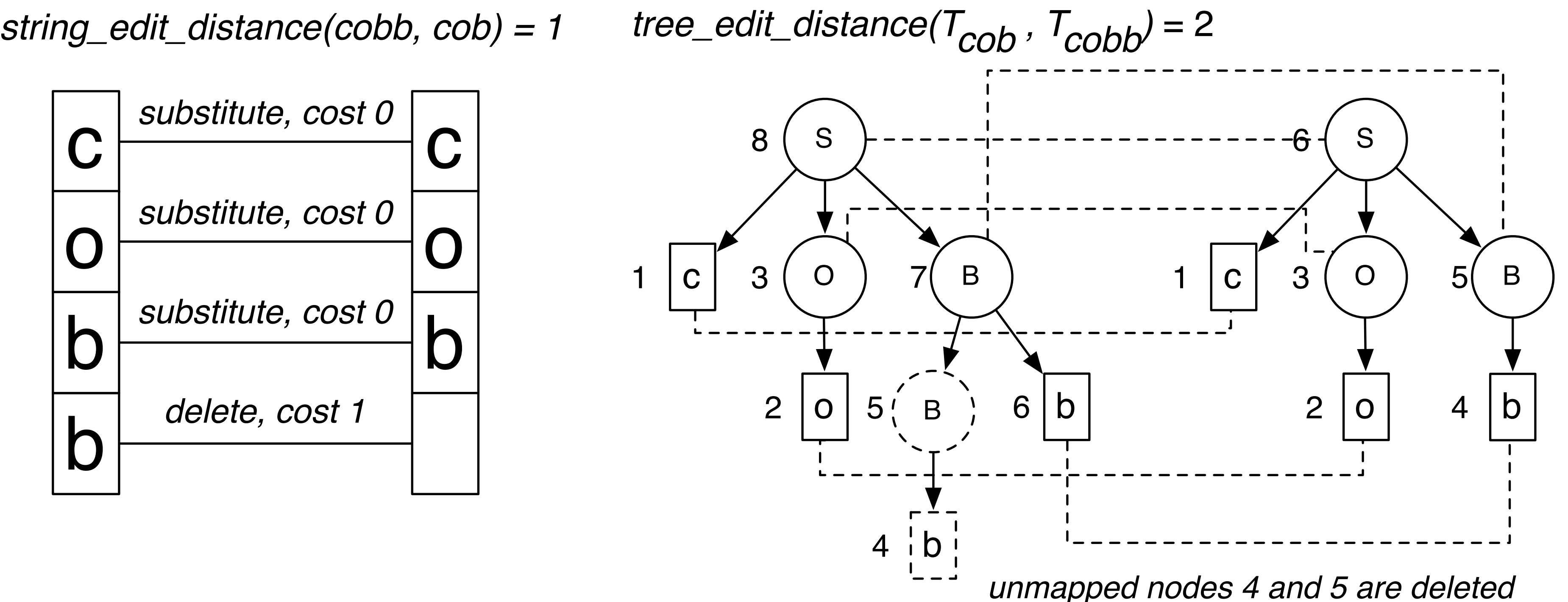


Figure 3.9: We can apply tree edit distance to the parse trees of strings in a context-free language. In this manner we account for changes to the string as well as changes to the language structures in the grammar by which the string was generated. In this example, we see that the string edit distance of *cobb* and *cob* is 1. In contrast, the tree edit distance of the parse trees for these two strings (T_{cobb} and T_{cob} respectively), is 2.

because we apply one fewer production in the derivation for *cob* than the derivation for *cobb* and we delete a leaf vertex for terminal *b*. In contrast to string edit distance, tree edit distance allows us to compare two strings relative to structures defined in a context-free grammar. When productions align with the specification of constructs in a high-level language (such as function definitions), a tree edit distance metric allows us to compare two strings in terms of these language constructs rather than the order of symbols in a string.

3.1.4 Section Summary

This section introduced concepts from language theory, parsing, and discrete mathematics that we use to both formalize security policy analysis (and text processing in general) as well as to directly address three core limitations of security policy analysis.

3.2 How We Address Limitations of Security-Policy Analysis

We will now demonstrate how we can use the concepts from theoretical computer science and mathematics introduced in the previous section in order to address our three core limitations of security policy analysis.

3.2.1 Policy Gap Problem

Figure 3.10 illustrates the gap between traditional text-processing tools and the languages used in security policies.

Many of the languages used in security policies are written in hierarchical object models. Hierarchical object models may contain recursion or arbitrarily deep hierarchies. For example PKI policies written in RFC 2527 and 3647 have a hierarchical

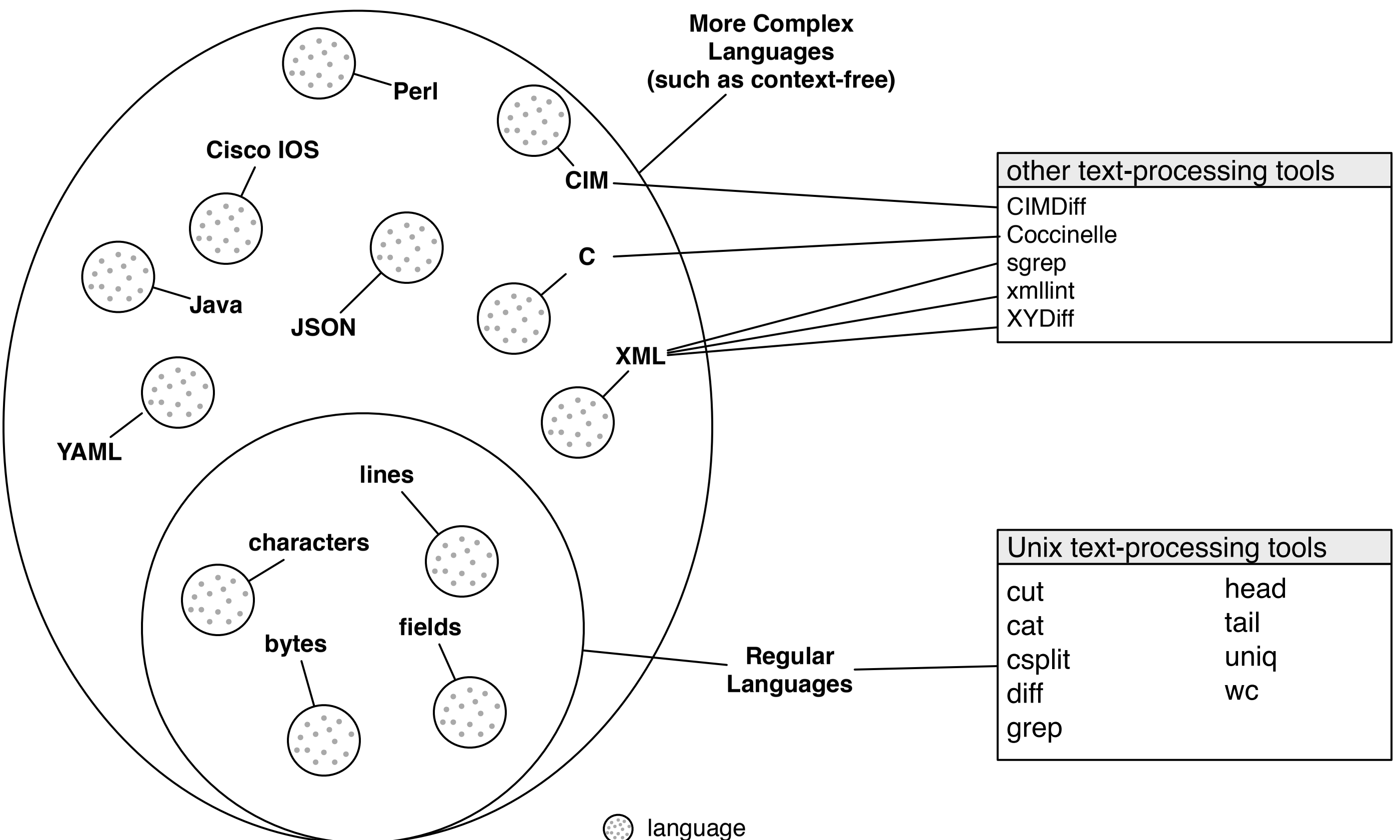


Figure 3.10: Although many high-level language constructs used by practitioners are non-regular, traditional `Unix` text-processing tools operate on regular languages. While specialized tools do exist to operate on other languages, they lack the generality of `Unix` text-processing tools. Our XUTools generalizes traditional `Unix` text-processing tools to a broader class of languages and this enables us to process and analyze many more high-level constructs.

set of provisions, configuration languages used for Cisco and Juniper devices have a hierarchical command language, and many data formats in the smart grid ranging from CIM, to IEC 61850's SCL and GOOSE have hierarchical object models.

Therefore, in order to process texts in terms of these models, tools need to be able to solve the parentheses-matching problem. We need to extend the class of languages that traditional Unix text-processing tools process beyond regular languages.

3.2.2 Granularity of Reference Problem

Grammars give us a natural way to formalize languages with hierarchical structure and this structure allows us to process a text at multiple levels of abstraction. Parse subtrees encode different ways to interpret a text with respect to the grammar whose start production is the production applied at the subtree root.

Natural-language legal documents (such as RFC 3647 PKI policies) illustrate this point. Figure 3.11 illustrates that if we write a grammar that aligns a parse tree with document structure, parse subtrees correspond to the entire policy, sections, and subsections.

A practitioner may interpret the same input text with respect to many different languages. For example, a network administrator may be interested in the set of interfaces defined on a router whereas an auditor of that same router may be interested in ACLs. Furthermore, these languages do not necessarily have to be in the same grammar.

In our research, we use parsing as a mechanism for practitioners to programmatically operate upon different *interpretations* of a source text where those interpretations correspond to languages defined in a context-free grammar.

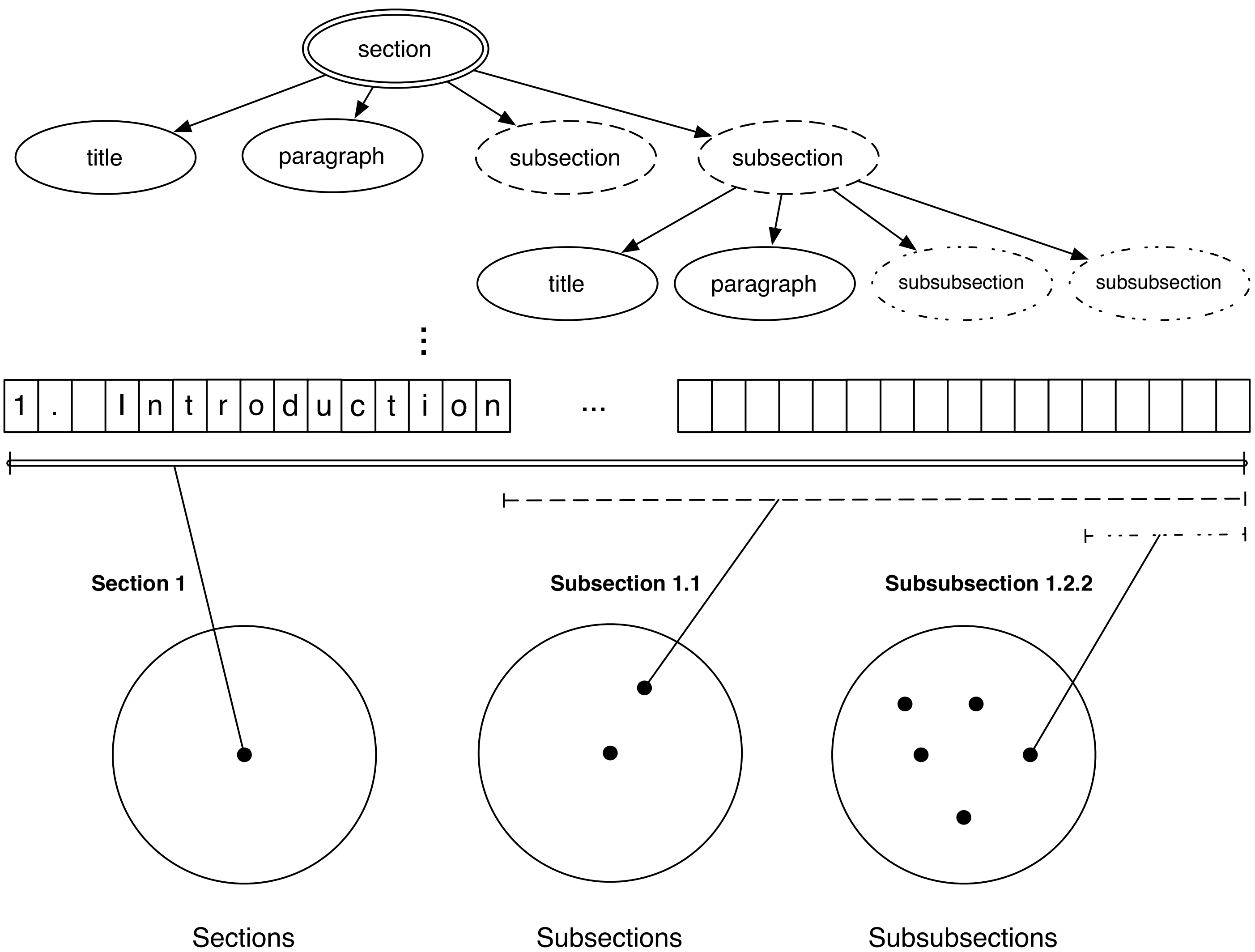
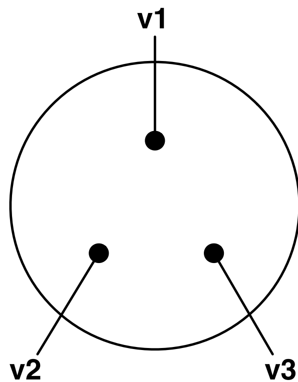


Figure 3.11: We can interpret a parse tree at multiple levels of abstraction. Here, we show how a grammar with productions for sections, subsections, and subsubsections may be used to analyze a PKI policy with respect to languages of sections, subsections, or subsubsections.



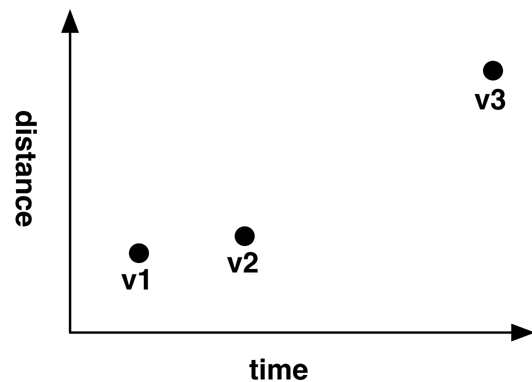


Figure 3.12: We can use string and tree edit distance metrics to measure trends in how security policies change over time. In this figure, we see that very little changed between versions 1 and 2 of the CERN Certificate Policy but that many changes occurred between versions 2 and 3.

3.2.3 Policy Discovery Needs Problem

If we view security policies and artifacts as strings in a language, then we can define operations upon these security primitives that give practitioners feedback for security policy.

During our research, we observed that we can use simple measures such as counting as well as string and tree distance metrics in order to compare and measure security primitives. When we have a language that contains multiple versions of the same policy artifacts, we can use distance metrics to measure how these artifacts evolve. Figure 3.12 illustrates that an EUGridPMA policy accreditation committee could measure how much a CA's policy has changed over time. The committee could extract the set of versioned PKI policies written by the CA under review, order those policies into a sequence by time, and then measure the distance between consecutive versions.

We should note that distance metrics between security primitives allow us to measure trends in the evolution of security and could eventually produce a geometry for structured text.

3.3 Conclusions

We introduced several concepts from language theory, parsing, and discrete mathematics in order to address the three core limitations of security policy analysis. In order to address the *tools gap problem*, we need to extend traditional text-processing tools from regular languages to context-free languages that can accommodate hierarchical object models found in modern policy formats. In order to address the *granularity of reference problem*, we can use parsing and parse trees as a mechanism to process text with respect to different languages at a variety of levels of abstraction. Finally, to address the *policy discovery needs problem*, we can use basic counting measures as well as string and tree distance metrics to quantify security-policy evolution. In the next chapter, we apply these theoretical underpinnings and employ our libxutools to solve real-world problems in network configuration management.

Chapter 4

Why and How to Use XUTools

In this chapter, we motivate our XUTools in the context of several real-world use cases and demonstrate how to use our tools.

4.1 XUTools and Real-World Use Cases

We now motivate each of our current XUTools (`xugrep`, `xuwc`, and `xudiff`) with real-world use cases that reveal some shortcomings of traditional Unix text-processing tools. We then provide detailed instructions of how to use our tools to solve problems related to network configuration management. Later, in Chapter 7, we apply our approach to give auditors new, practical capabilities to measure the security posture of their networks.

4.1.1 XUGrep

Traditional Unix `grep` extracts all lines in a file that contain a substring that match a regular expression.

`xugrep` generalizes the class of languages that we can practically extract in one command from regular to context free. Figure 4.1 shows the command-line syntax.

xugrep reports all strings in the input files that satisfy an xupath query. At a high-level an xupath consists of a sequence of references to language structures for xugrep to extract. The role of xupaths will become more apparent in the examples to follow. Finally, when present, the --R2=LE option causes xugrep to output a table where each row contains a match for the xupath query.

xugrep usage
xugrep [--R2=LE] <xupath> <input_file>+

Figure 4.1: Our xugrep reports all strings that satisfy the xupath within the context of the input files. The --R2=LE option reports a table of matches in which each row corresponds to a match for the xupath query.

.

During the design of xugrep, practitioners from several domains suggested use cases for this tool. We now motivate xugrep with some of these real-world examples.

Network Configuration Management

The prevalence of multi-line, nested-block-structured formats in network configuration management has left a capability gap for traditional tools. The configuration languages for Cisco and Juniper network devices for example, are both hierarchically-structured languages. Figure 4.2 shows a fragment of a Cisco IOS configuration file in which two network interfaces are defined.

```
router.v1.example
interface Loopback0
 description really cool description
 ip address 333.444.1.185 255.255.255.255
 no ip unreachables
 ip pim sparse-dense-mode
 crypto map azalea
!
interface GigabitEthernet4/2
 description Core Network
 ip address 444.555.2.543 255.255.255.240
 ip access-group outbound_filter in
 ip access-group inbound_filter out
 no ip redirects
 no ip unreachables
 no ip proxy-arp
!
```

Figure 4.2: Cisco IOS has a hierarchically-structured configuration syntax. This fragment consists of two interface blocks which contain five and seven lines respectively.

Currently, if practitioners want to extract interfaces from a Cisco IOS router configuration file, they may craft an invocation for `sed`

```
sed -n '/^interface
  ATM0/,/^!/\{/^\!d;p;\}' router.v1.ios
```

In contrast, with `xugrep` practitioners only have to type

```
xugrep --R2=LE '//ios:interface' router.v1.ios
```

Figure 4.3 shows the output of `xugrep` for the above command. Given the input file `router.v1.ios`, the xupath query (//IOS:INTERFACE) tells `xugrep` to extract all interface blocks.

More formally, `xugrep` extracts all strings in the language of the *interface* production in the Cisco IOS grammar (IOS). `xugrep` outputs a table where each row corresponds to a match. The first column contains the file within which the match

xugrep Cisco IOS examples

```
bash-ios-ex1$ xugrep --R2=LE "//ios:interface" router.v1.example

FILE                IOS:INTERFACE       TEXT
router.v1.example  Loopback0           interface Loopback0\n description really cool description\n ip
address 333.444.1.185 255.255.255.255 no ip redirects\n no ip unreachables\n ip pim sparse-dense-mode\n
crypto map azalea\n!

router.v2.example  GigabitEthernet4/2  interface GigabitEthernet4/2\n description Core Network\n ip
address 444.555.2.543 255.255.255.240 \n ip access-group outbound_filter in\n ip access-group
inbound_filter out\n no ip redirects\n no ip unreachables\n no ip proxy-arp\n!

bash-ios-ex2$ xugrep "//ios:interface" router.v1.example

interface Loopback0
  description really cool description
  ip address 333.444.1.185 255.255.255.255 no ip redirects
  no ip unreachables
  ip pim sparse-dense-mode
  crypto map azalea
!
interface GigabitEthernet4/2
  description Core Network
  ip address 444.555.2.543 255.255.255.240
  ip access-group outbound_filter in
  ip access-group inbound_filter out
  no ip redirects
  no ip unreachables
  no ip proxy-arp
!
```

Figure 4.3: Practitioners may use `xugrep` to extract high-level language constructs such as interfaces. These two example invocations demonstrate two possible usage scenarios depending upon whether extracted blocks should be displayed on a single line or not.

occurred, the second column holds the name of the interface, and the final column contains the block with escaped newlines.

C Source Code

Practitioners may want to be able to recognize (and thereby extract) all C function blocks in a file. As stated by one person on Slashdot following our *LISA* 2011 poster presentation, "it would be nice to be able to grep for a function name as a function name and not get back any usage of that text as a variable or embedded in a string, or a comment" [107, 142][1]. Furthermore, practitioners may also want to extract function blocks or calls to a particular function relative to the function in which it occurs. Figure 4.4 shows a simple C source file in which five functions are defined.

c.v1.example

```
int
putstr(char *s)
{
  while(*s) {
    putchar(*s++);
  }
}

int
fac(int n)
{
  if (n == 0) {
    return 1;
  } else {
    return n*fac(n-1);
  }
}
```

```
// Comment on putn right here.
int
putn(int n)
{
  if (9 < n){
    putn(n / 10);
  }
  putchar((n%10) + '0');
}

int
facpr(int n)
{
  putstr("factorial ");
  putn(n);
  putstr(" = ");
  putn(fac(n));
  putstr("\n");
}
```

```
int
main()
{
  int i;
  i = 0;
  while(i < 10){
    facpr(i++);
  }
  return 0;
}
```

Figure 4.4: The C programming language uses matching parentheses to specify blocks for a variety of constructs that include functions and conditional statements. The example file in this figure contains five function definitions.

Traditional grep cannot handle these use cases because it requires us to solve the parentheses-matching problem. We need a tool that can recognize blocks in order to extract function blocks or function calls within a function block.

[1]Thanks to Tanktalus for the Slashdot post.

grep and xugrep C examples

```
bash-c-ex1$ grep -n "putn" c.v1.example

20:// Comment on putn right here
22:putn(int n)
25:    putn(n / 10);
30:// Facpr calls putn FYI
35:    putn(n);
37:    putn(fac(n));

bash-c-ex2$ xugrep --R2=LE "//cspec:function" c.v1.example

FILE         CSPEC:FUNCTION  TEXT
c.v1.example putstr          int\nputstr(char *s)\n{\n while(*s) {\n   putchar(*s++);\n  }\n}

c.v1.example fac             int\nfac(int n)\n{\n if (n == 0){\n  return 1;\n } else {\n return n*fac(n-1);
\n  }\n}

c.v1.example putn            int\nputn(int n)\n{\n  if(9 < n) {\n  putn(n / 10);\n  }\n  putchar((n%10) +
'0');\n}

c.v1.example facpr           int\nfacpr(int n)\n{\n  putstr("factorial ");\n  putn(n);\n  putstr(" = ");\n
putn(fac(n));\n  putstr("\n");\n}

c.v1.example main            int\nmain()\n{\n  int i;\n  i=0;\n  while(i < 10){\n  facpr(i++);\n  }\n return
0;\n}

bash-c-ex3$ xugrep --R2=LE "//cspec:function/builtin:line[re:testsubtree('putn','e')]" c.v1.example

FILE         CSPEC:FUNCTION  BUILTIN:LINE   TEXT
c.v1.example putn            2              putn(int n)
c.v1.example putn            5              putn(n / 10);
c.v1.example facpr           5              putn(n);
c.v1.example facpr           7              putn(fac(n));
```

Figure 4.5: The C programming language contains parenthesis-delimited blocks of code nested at arbitrary depths. The first example (`bash-c-ex1`) shows how `grep` may be used to extract occurrences of function names. However, practitioners may also want to extract function blocks (`bash-c-ex2`) or lines relative to function blocks (`bash-c-ex3`) and these are both handled by `xugrep`.

The first example of Figure 4.5 (`bash-c-ex1`) shows that `grep` does not distinguish between comments and calls within function blocks when we extract lines that contain *putn* from the input of Figure 4.4.

Function blocks are not regular because brackets close C functions but those functions may contain other kinds of blocks (such as if-statements) which are similarly delimeted by parentheses. Without parentheses-matching, the closing brackets for these constructs are ambiguous.

The second example of Figure 4.5 (`bash-c-ex2`) shows how we can use `xugrep` to extract all function blocks from the input file shown in Figure 4.4. Given the input file, `c.v1.example`, the xupath query tells `xugrep` to extract all strings that are in the language of the *function* production in the C grammar (CSPEC). The `--R2=LE` flag outputs a table of matches. Each row in the table contains three columns, the name of the file processed, the *label* of the function block extracted, and the contents of the function block (with newlines escaped).

The third example of Figure 4.5 (`bash-c-ex3`) shows a more involved query with `xugrep` that extracts all lines that contain `putn` within the context of a function block. Given the input file `c.v1.example`, the xupath query tells `xugrep` to first extract all function blocks (specified by the *function* production in the grammar for C (`cspec`). Then, having extracted the functions, the next step in the xupath (each step is separated by the `/` delimiter) tells `xugrep` to extract each line within the function. Finally, the predicate `[re:testsubtree('putn','e')]` tells `xugrep` to only output lines that contain the string *putn*. Again, the `--R2=LE` flag outputs matches in a table where each row is a match. The first column reports the file from which matches were extracted. The second and third columns report the function name and line number for each match. The final column contains the match.

NVD-XML:

Practitioners at the RedHat Security Response Team wanted a way to process the XML feed for the National Vulnerability Database (NVD).[2] Specifically, they wanted to know how many NVD entries contained the string `cpe:/a:redhat`, the vulnerability score of these entries, and how many XML elements in the feed contain `cpe:/a:redhat`.

Traditional grep cannot handle this use case because it requires us to solve the parentheses-matching problem. This limitation motivates the capability to be able to report matches with respect to a given context.

In contrast, `xugrep` that can handle strings in context-free languages because when we extract XML elements, we must asociate opening and closing tags. Multiple XML elements may share the same closing tag, and XML elements may be nested arbitrarily deep. Therefore, we need parentheses matching to recognize XML elements. Moreover, we need a `grep` that can report matches with respect to the contexts defined within the NVD-XML vocabulary. (Although `xmllint`'s shell-mode grep certainly provides one solution, it is not general enough to deal with languages other than XML [145]. We will compare `xmllint` to our own tools in more detail in Chapter 6.)

4.1.2 XUWc

As stated by the `Unix` man pages, traditional `wc` counts the number of words, lines, characters, or bytes contained in each input file or standard input [110].

`xuwc` generalizes `wc` to count strings in context-free languages and to report those counts relative to language-specific constructs. Figure 4.6 shows the command-line syntax for `xuwc`.

[2]http://nvd.nist.gov/download.cfm#CVE_FEED

```
xuwc usage
xuwc [--count=<grammar:production>]
     [--context=<grammar:production>]
     <xupath> <input_file>+
```

Figure 4.6: Given an xupath and a set of files, `xuwc` will count all matches in the result corpus.

We now discuss some real-world examples that motivate `xuwc`.

Network Configuration Management

Network administrators configure and maintain networks via language-specific constructs, such as interfaces, and they would like to be able to get statistics about their configuration files in terms of these constructs. Administrators might like to measure the number of interfaces per router, or even the number of lines or bytes per interface. For example, one network administrator at Dartmouth Computing Services wanted to know how many campus router interfaces use a particular Virtual Local Area Network (VLAN). Figure 4.7 shows two versions of the same Cisco IOS configuration file that we will use in our running examples.

Traditional wc cannot handle this use case. Currently, `wc` lets practitioners count the number of bytes, characters, lines, or words within a file. Administrators and auditors can use `wc` to calculate lines of configuration associated with a network device.

Figure 4.8, Example 1 (`bash-ios-ex1`) shows how an administrator can count the number of lines in a configuration file and example two (`bash-ios-ex2`) shows how to estimate the number of interfaces in a configuration by pipelining the output of `grep` into `wc`. In the first example, `wc` takes two files as input (`router.v1.example` and `router.v2.example`) and the `-l` flag that tells `wc` to output the number of lines in each file as well as the total number of lines. In the second example, we can

router.v1.example	router.v2.example

`interface Loopback0`
` description really cool description`
` ip address 333.444.1.185 255.255.255.255`
` no ip unreachables`
` ip pim sparse-dense-mode`
` crypto map azalea`
`!`
`interface GigabitEthernet4/2`
` description Core Network`
` ip address 444.555.2.543 255.255.255.240`
` ip access-group outbound_filter in`
` ip access-group inbound_filter out`
` no ip redirects`
` no ip unreachables`
` no ip proxy-arp`
`!`
	`interface Loopback0`
	` description really cool description`
	` ip address 333.444.1.581 255.255.255.255`
	` no ip unreachables`
	` ip pim sparse-dense-mode`
	` crypto map daffodil`
	`!`
	`interface GigabitEthernet4/2`
	` description Core Network`
	` ip address 444.555.2.543 255.255.255.240`
	` ip access-group outbound_filter in`
	` no ip redirects`
	` no ip unreachables`
	` no ip proxy-arp`
	` ip flow ingress`
	`!`

Figure 4.7: Network administrators want to count the number of high-level language constructs within a set of files. For example, administrators may want to compare two versions of a configuration file by counting language structures at different levels of abstraction.

extract the number of `interface` commands at the start of an interface block via the regular expression `interface`. Since `grep` outputs one line per match, and each match corresponds to an interface, we can pipe the output to `wc -l` to count the number of interfaces in a file.

Example 3 (`bash-ios-ex3`) in Figure 4.8 illustrates how practitioners can use `xuwc` to *directly* count the number of interfaces in network device configurations. In the previous example, we could indirectly count the number of interfaces by counting the number of times an interface block was opened. Unlike `grep` pipelined with `wc`, `xuwc` can count structures in a context-free language. Given input files `router.v1.example` and `router.v2.example` the xupath //IOS:INTERFACE tells `xuwc` to output the number of interface blocks in both input files.

We could make `wc` partially-aware of context-free languages by piping the output of `xugrep` to `wc`. For example, we could use our `xugrep` to extract the interfaces in the configuration in document order and escape the newlines in each block (via the `--R2=LE` option). Each line in `xugrep`'s output would correspond to an interface in the configuration files. We could then redirect this output to `wc -l` to count the

wc and xuwc Cisco IOS examples

```
bash-ios-ex1$ wc -l router.v1.example router.v2.example
50 router.v1.example
49 router.v2.example
99 total

bash-ios-ex2$ grep "^interface" router.v1.example router.v2.example | wc -l
4

bash-ios-ex3$ xuwc "//ios:interface" router.v1.example router.v2.example

COUNT   COUNT UNIT      CONTEXT
2       ios:interface   router.v1.example
2       ios:interface   router.v2.example
2       IOS:INTERFACE   2       FILE      TOTAL

bash-ios-ex4$ xuwc "//ios:interface/builtin:line" router.v1.example router.v2.example

COUNT   COUNT UNIT      CONTEXT
16      builtin:line    router.v1.example
16      builtin:line    router.v2.example
32      BUILTIN:LINE    2       FILE      TOTAL

bash-ios-ex5$ xuwc --context=ios:interface "//ios:interface/builtin:line" router.v1.example
  router.v2.example

COUNT   COUNT UNIT      CONTEXT
7       builtin:line    router.v1.example,Loopback0
9       builtin:line    router.v1.example,GigabitEthernet4/2
7       builtin:line    router.v2.example,Loopback0
9       builtin:line    router.v2.example,GigabitEthernet4/2
32      BUILTIN:LINE    4       IOS:INTERFACE   TOTAL
```

Figure 4.8: Network administrators and auditors might use `wc` and `grep` to count high-level structures within a file but `xuwc` lets practitioners count relative to non-file contexts.

number of lines. Since lines correspond to interfaces, `wc` would give us the number of interfaces in the configuration files.

A pipeline of `xugrep` and `wc` does not allow practitioners to easily count structures relative to a non-file context. For example, `wc` always reports the number of bytes, characters, lines, or words in the context of the input files. Practitioners may want to count structures that can't be recognized by a regular expression and report those counts relative to a non-file context. Examples 4 and 5 in Figure 4.8 demonstrate how to use `xuwc` to count the number of lines per interface and report results in two different ways. Given our two example router input files, the xupath //IOS:INTERFACE/BUILTIN:LINE tells `xuwc` to count the number of lines contained within each interface. In Example 4, `xuwc` outputs the number of lines per interface in the context of the entire file. In contrast, Example 5 uses the `--context` flag so that `xuwc`outputs counts in the context of interfaces. For example, we can see that the `GigabitEthernet4/2` interface in `router.v1.example` contains 9 lines.

4.1.3 XUDiff

Traditional Unix `diff` computes an edit script between two files in terms of their lines. `diff` outputs an edit script that describes how to transform the sequence of lines in the first file into the sequence of lines in the second file via a sequence of edit operations (delete, insert, substitute) [39]. All of these edit operations are performed upon *lines* in the context of the *entire* file.

While traditional `diff` lets practitioners compare files in terms of the line, our `xudiff` allows practitioners to compare files in terms of higher-level language constructs specified by a context-free grammar. Figure 4.9 shows the command-line syntax for `diff` and `xudiff` respectively.

```
xudiff usage
xudiff [--cost=<cost_fn>] <xupath> <input_file1> <input_file2>
```

Figure 4.9: Our `xudiff` compares two files in terms of the parse trees generated by applying an xupath to each file. An optimal edit-cost function affects the choice among competing matches.

We designed `xudiff` with real-world use cases in mind. We now motivate `xudiff` with a few of those usage scenarios.

Document-Centric XML

A wide variety of documents ranging from webpages, to office documents, to digitized texts are encoded according to some XML schema. Practitioners may want to compare versions of these documents in terms of elements of that schema. For example, a security analyst may want to compare versions of security policies in terms of sections, or subsections. Although tools exist to compare XML documents [30, 145], we offer a general-purpose solution for a wider variety of structured texts.

Network Configuration Management

Current tools such as the Really Awesome New Cisco config Differ (RANCID) [103] let network administrators view changes to router configuration files in terms of lines. However, administrators may want to view changes in the context of other structures defined by Cisco IOS. Alternatively, network administrators may want to compare configurations when they migrate services to different routers.

If a network administrator moves a network interface for a router configuration file, then a line-based edit script for the router configurations may report the change as 8 inserts and 8 deletes. However, an edit script that reports changes in terms of interfaces terms of interfaces ("interface X moved") might be more readable and less

computationally intensive.

Example 1 (`bash-ios-ex1`) of Figure 4.10 shows the edit script when an administrator runs `diff` on the two files in Figure 4.7. The `diff` command takes two input files and outputs the line-level differences between them. In the resultant output, we see that the `ip address` was changed (the IP address 333.444.1.185 was changed to 333.444.1.581). The `crypto map azalea` line was changed to `crypto map daffodil`. The `ip access-group inbound_filter out` line was deleted and `ip flow ingress` was inserted.

Practitioners, however, may want to be able to summarize changes between two configuration files at arbitrary levels of abstraction as represented by the Cisco IOS language. *Traditional diff cannot handle this use case* because it requires us both to solve the parentheses-matching problem, and to process and report changes relative to the context encoded by the parse tree.

Although the full Cisco IOS grammar is context-sensitive, meaningful subsets of the grammar, such as interface blocks and other nested blocks, are context-free [16]. Before we can compare interface blocks, we need to be able to easily extract them.

In this use case, we are interested in how the sequence of interface blocks changed between two versions of a router configuration file. If we wanted only to understand how the sequence of lines or sequence of interfaces changed, then we could use our `xugrep` to extract the interfaces or lines in document order, escape the newlines in each block, and pipe the sequence of interfaces or lines into `diff`.

However, we want to understand how the lines in a configuration change with respect to the contexts defined by the Cisco IOS language. We want to report changes in terms of the entire configuration file or even in terms of individual interfaces. Examples 2-4 in Figure 4.10 and Figure 4.11 illustrate how we can use `xudiff` to report changes at different levels of abstraction.

Example 2 (`bash-ios-ex2`) shows how practitioners can pipeline `xudiff` with

diff and xudiff IOS examples

```
bash-ios-ex1$ diff -u router.v1.example router.v2.example
--- data/test/cisco_ios/router.v1.example       2013-01-04 07:00:53.000000000 -0600
+++ data/test/cisco_ios/router.v2.example       2012-12-16 00:29:44.000000000 -0600
@@ -1,16 +1,16 @@
 interface Loopback0
   description really cool description
-  ip address 333.444.1.185 255.255.255.255 no ip redirects
+  ip address 333.444.1.581 255.255.255.255 no ip redirects
   no ip unreachables
   ip pim sparse-dense-mode
-  crypto map azalea
+  crypto map daffodil
 !
 interface GigabitEthernet4/2
   description Core Network
   ip address 444.555.2.543 255.255.255.240
   ip access-group outbound_filter in
-  ip access-group inbound_filter out
   no ip redirects
   no ip unreachables
   no ip proxy-arp
+  ip flow ingress
 !

bash-ios-ex2$ xudiff "//ios:config" router.v1.example router.v2.example | egrep "ios:config"
U       4       0       root (ios:config)                                   ===> root (ios:config)

bash-ios-ex3$ xudiff "//ios:config" router.v1.example router.v2.example | egrep "ios:config"
U       4       0       root (ios:config)                                   ===> root (ios:config)
U       2       0       Loopback0 (ios:interface)                           ===> Loopback0 (ios:interface)
U       2       0       GigabitEthernet4/2 (ios:interface)                  ===> GigabitEthernet4/2 (ios:interface)
```

Figure 4.10: Our `xudiff` compares two files in terms of their parse trees and so practitioners may view changes at multiple levels of abstraction. In contrast, `diff` operates on one level of abstraction, the line.

xudiff IOS examples

```
bash-ios-ex4$ xudiff "//ios:config" router.v1.example router.v2.example
U       4       0       root (ios:config)                                  ===> root (ios:config)
U       2       0       Loopback0 (ios:interface)                          ===> Loopback0 (ios:interface)
U       0       0       description really cool description (builtin:line) ===> description really cool description
(builtin:line)
U       1       1       ip address 333.444.1.185 255.255.255.255 no ip redirects (builtin:line) ===> ip address
333.444.1.581 255.255.255.255 no ip redirect\
s (builtin:line)
U       0       0       no ip unreachables (builtin:line)                  ===> no ip unreachables (builtin:line)
U       0       0       ip pim sparse-dense-mode (builtin:line)            ===> ip pim sparse-dense-mode
(builtin:line)
U       1       1       crypto map azalea (builtin:line)                   ===> crypto map daffodil (builtin:line)
U       2       0       GigabitEthernet4/2 (ios:interface)                 ===> GigabitEthernet4/2 (ios:interface)
U       0       0       description Core Network (builtin:line)            ===> description Core Network
(builtin:line)
U       0       0       ip address 444.555.2.543 255.255.255.240 (builtin:line) ===> ip address 444.555.2.543
255.255.255.240 (builtin:line)
U       0       0       ip access-group outbound_filter in (builtin:line)  ===> ip access-group outbound_filter in
(builtin:line)
D       1       1       ip access-group inbound_filter out (builtin:line)  ===> ^
U       0       0       no ip redirects (builtin:line)                     ===> no ip redirects (builtin:line)
U       0       0       no ip unreachables (builtin:line)                  ===> no ip unreachables (builtin:line)
U       0       0       no ip proxy-arp (builtin:line)                     ===> no ip proxy-arp (builtin:line)
I       2       1       ^                                                  ===> ip flow ingress (builtin:line)

bash-ios-ex5$ xudiff  --cost_fn=word_edist_cost "//ios:config" router.v1.example router.v2.example
| egrep "ios:interface|ios:config"
U       9       0       root (ios:config)                                  ===> root (ios:config)
U       2       0       Loopback0 (ios:interface)                          ===> Loopback0 (ios:interface)
U       7       0       GigabitEthernet4/2 (ios:interface)                 ===> GigabitEthernet4/2 (ios:interface)

bash-ios-ex6$ xudiff  --cost_fn=char_edist_cost "//ios:config" router.v1.example router.v2.example
| awk '$3 < 5 {print $0;}' | awk '$3 > 0 { print $0;}'
U       2       2       ip address 333.444.1.185 255.255.255.255 no ip redirects (builtin:line) ===> ip address
333.444.1.581 255.255.255.255 no ip redirects (builtin:line)
```

Figure 4.11: In this Figure, we see more examples of `xudiff` in action. Specifically, we see the ability to specify a cost function between parse tree nodes. As a result, we can isolate changes that are very subtle within multiple versions of the configuration files.

egrep to get a summary of changes to the entire router configuration. Given two input files and the xupath of IOS:CONFIG, xudiff parses both files relative to the grammar production for a Cisco IOS configuration file and then compares the resultant parse trees with a tree edit distance algorithm (we introduced tree edit distance algorithms in Chapter 3). We then use egrep to extract the portion of the edit script that applies to the entire configuration file. If we read the output, we see that the subtree for the entire router.v1.example was updated (U) to the subtree for the entire router.v2.example. The cost of updating the root nodes for both configurations' parse trees was 0, but the accumulated cost of updating the nodes within each parse tree was 4. By default, xudiff assigns a unit cost to delete, insert, or update a node in a parse tree.

If practitioners want to understand changes to the configuration file in terms of interfaces, then they can modify the egrep command to extract lines in the edit script that correspond to interface blocks. The modified pipeline in Example 3 (bash-ios-ex3) shows that the Loopback0 interface as well as the GigabitEthernet4/2 interface blocks were modified. Each of those blocks corresponds to a subtree in the parse trees for the input files. The cost to modify the root node of each interface's subtree was 0, but the overall cost was 2 per interface.

Alternatively, practitioners may want to view all changes, Example 4 (bash-ios-ex4) shows the entire edit script. One thing that we might notice is that an IP address and a single word were updated in the Loopback0 interface but that a line was deleted and inserted in the GigabitEthernet4/2 interface. Our edit script in Example 3 however, reported the same amount of changes in both interface blocks (2 nodes per interface subtree). xudiff allows practitioners to use different edit costs so as to bring out changes below the level of individual parse tree nodes.

In Example 5 (bash-ios-ex5) xudiff uses the --cost_fn flag to compare parse tree node labels using word edit distance. As a result, we can see that in terms

of words, GigabitEthernet4/2 changed much more (7 words) than Loopback0 (2 words).

Finally, practitioners may want to find changes between versions of a file that are very subtle, perhaps differing by just a few characters. Example 6 (bash-ios-ex6) shows how to use xudiffwith a character-based cost function combined with awk, to filter out nodes in the parse tree whose labels changed by 1–4 characters.

4.2 Conclusions

Our XUTools to address practical use cases that current text-processing tools can not. During the design process of our tools, we spoke with system administrators, auditors, and developers to understand the full spectrum of use cases for Unix tools that operate on context-free languages. Although our approach has a language-theoretic foundation, we hope that this chapter has demonstrated the practical implications for XUTools. In the next chapter (Chapter 5), we will describe the design and implementation of our XUTools.

Chapter 5

Design and Implementation of XUTools

The security problems we encountered in the fieldwork of Chapter 2 reduce to the need to efficiently manipulate and analyze structured text—like Unix did years ago, but for a broader class of languages. We designed XUTools to extend the class of languages upon which practitioners can practically compute.

In the first section of this chapter, we reinterpret each of the three core limitations of security policy analysis as design requirements for text-processing tools and sketch how our XUTools meet those requirements.

In the second section of this chapter, we describe the implementation of XUTools.

5.1 Design Requirements

The three core limitations of security policy analysis suggest design requirements for XUTools.

5.1.1 Tools Gap Problem

We designed our text-processing tools to address the capability gap between traditional text-processing tools and the policy languages we encountered during our fieldwork. Traditional Unix tools primarily operate on regular languages–sets of strings that are in the language of a regular expression. If a string is in the language of a regular expression, then we say that the string *matches* that regular expression. Unix tools also operate by splitting an input string wherever a match occurs. Consider the following examples:

1. We can use a regular expression to match characters in an input string. According to the `grep` man page, these are the most fundamental expressions because they are the building blocks upon which other regular expressions are built [56]. The *Portable Operating System Interface for uniX (POSIX)* defines certain classes (or sets) of characters that are routinely used to build expressions [99]. These character classes include alphanumeric characters, lower case characters, punctuation, and whitespace.

2. We can use regular expressions to process a sequence of elements separated by a character called a *delimiter*. For example, we can view the contents of an input file as a sequence of lines. The delimiter for elements in this sequence is the newline character. We can write a simple regular expression to search for all matches of the newline character in the input string. The strings of text between matches correspond to lines.

3. We can use regular expressions to process fields in a *Comma-Separated Value (CSV)* file. CSV files encode tabular data where each row corresponds to a line. Lines consist of a sequence of fields that are delimited by a comma. We can use regular expressions to iterate through items in this table in row-major order. First, we can extract the lines in the manner described in the previous bullet.

For each line, we can search for all matches of the comma character and the strings of text between matches correspond to fields.

When lines and files correspond to meaningful constructs in markup, configuration, and programming languages, traditional `Unix` tools work well. For example, each line of an Apache web server's log file corresponds to an HTTP request received by the server.

Many file formats found in markup, configuration, and programming languages use hierarchical object models. Hierarchical object models may contain arbitrarily deep hierarchies. Our `xugrep`, `xuwc`, and `xudiff` allow practitioners to extract, count, and compare texts in terms of these hierarchical object models and thereby address the *Tools Gap Problem*.

5.1.2 Granularity of Reference Problem

Second, practitioners need to be able to process texts on multiple levels of abstraction. In Chapter 3, we observed that grammars and parse trees give us a natural way to formalize languages with hierarchical structures that correspond to different levels of abstraction. Therefore, we designed our XUTools to operate upon parse trees.[1]

5.1.3 Policy Discovery Needs Problem

Third and finally, practitioners need to measure security policies and how they change. Therefore, formalized policy analysis techniques in terms of the datatype operations discussed in Chapter 3. Our `xuwc` and `xudiff` tools both allow practitioners to count and compare evolving security primitives using measures such as counting high-level

[1]We should note that in addition to "eXtended Unix", we also chose the *xu* prefix from the Ancient Greek word ξυλον, denoting "tree" or "staff". We find the former sense of the word especially appropriate for the second design requirement given that XUTools operate on parse trees and process texts relative to languages with hierarchical structure.

structures as well as string and tree distance metrics.[2]

5.2 XUTools Internals

We now discuss the internals of our XUTools. For each tool we will provide a detailed working example and then discuss that tool's algorithm and implementation.

5.2.1 XUGrep Internals

As mentioned in Chapter 4, `xugrep` reports all strings in the input files that satisfy an xupath query. We now explain how `xugrep` works in detail.

Working Example

We now provide a detailed example of how `xugrep` extracts high-level language constructs from one or more input files. We focus on a call to `xugrep` that extracts all lines contained within each interface block from the configuration file shown in Figure 4.2 of Chapter 4.

As described in Chapter 4, `xugrep` takes one or more input files and a xupath as input and extracts all strings in the language of the xupath. In Figure 5.1 we see that `xugrep` first parses the xupath into a xupath parse tree using a grammar that specifies xupath. `xugrep` interprets an xupath as a tree that encodes an iterative querying process, a *xupath query tree*. The query tree root's children correspond to the contents of each input file. We view the leaf nodes of the query tree as a corpus, or a set of strings in the language at the leaf-level of the query tree. Here, the leaf-level corresponds to all strings in the language of the IOS:CONFIG production.

Our `xugrep` processes a query via a postorder traversal of the xupath parse tree.

[2]We should note that the sense of the word ξυλον as "staff" is appropriate for this third design requirement libxutools support administrators and auditors by automating currently-manual analysis techniques, allowing them to focus on higher-level change trends and more complicated analyses.

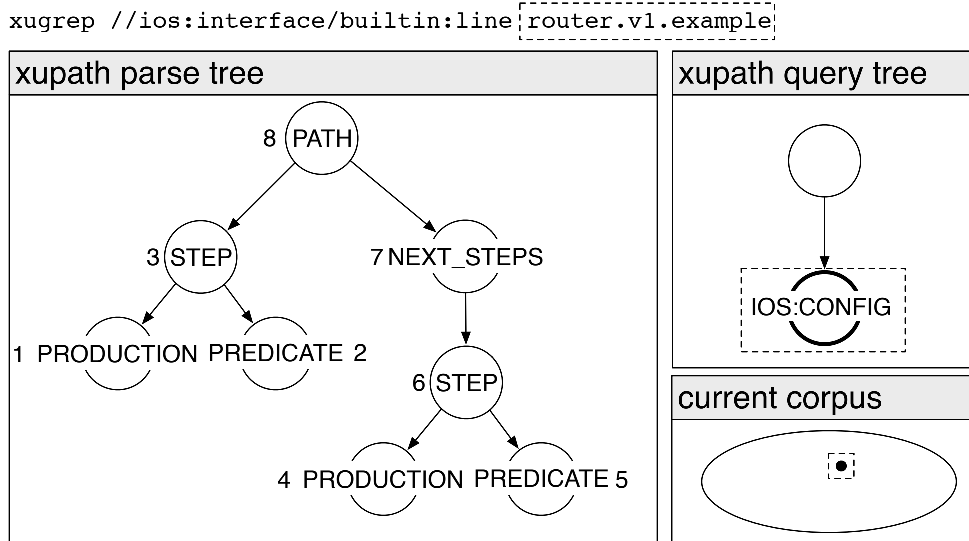
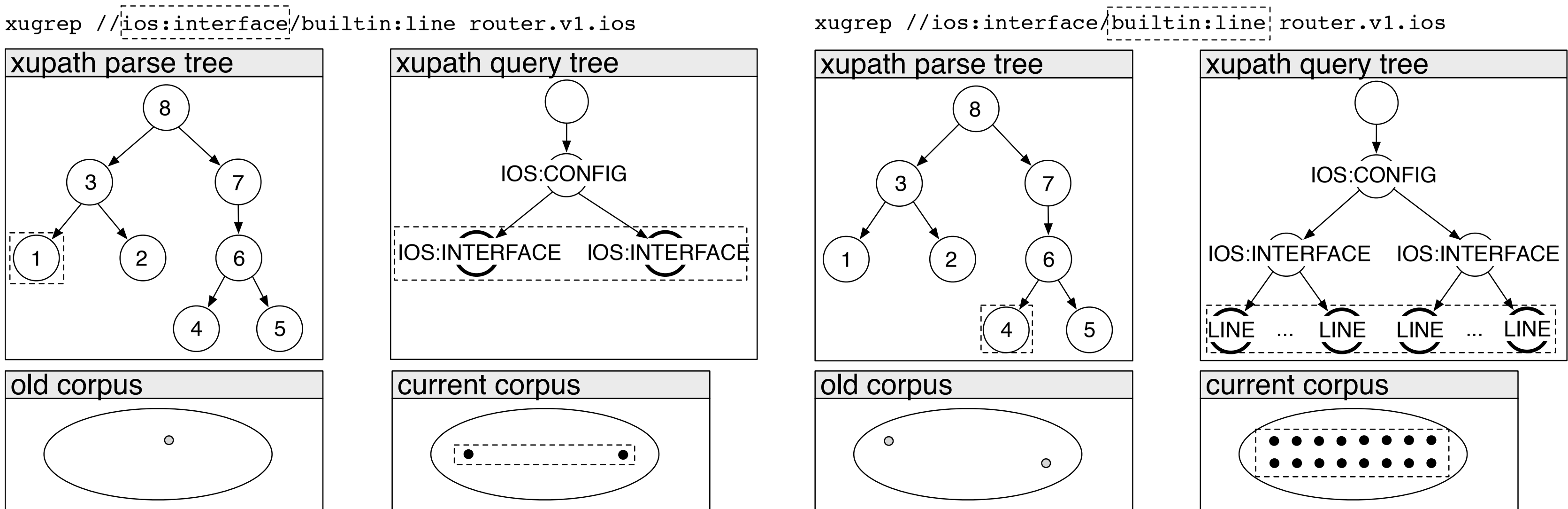


Figure 5.1: In the command above, we invoke `xugrep` on a router configuration file to extract all lines contained within interfaces. Specifically, `xugrep` parses the xupath (//IOS:INTERFACE/BUILTIN:LINE into a xupath parse tree. In addition, `xugrep` initializes a xupath query tree and a corpus.

During the traversal when a node corresponding to a production has been reached, `xugrep` extracts all strings in the language of that production name from the (old) corpus. Figure 5.2 shows that when `xugrep` reaches node 1 in the xupath parse tree, that it extracts all strings in the language of the IOS:INTERFACE production from the (old) corpus. These matching strings are then used to construct the next level of the xupath query tree and the leaves of this tree form the (new) current corpus. Figure 5.3 shows a similar process to extract the lines contained within each interface. Once the entire xupath parse tree has been traversed, `xugrep` outputs the current corpus.

We should note however, that the structure of the xupath query tree and its vertices, allow us to report the result set relative to different levels of context. Specifically, we can report the elements in the result set relative to the entire input corpus by outputting the labels of the tree vertices on the path from the result corpus element to the tree root.

Traditional `grep` reports matching lines relative to the context of the file in which the match was found. In contrast, our `xugrep` reports matching corpus elements



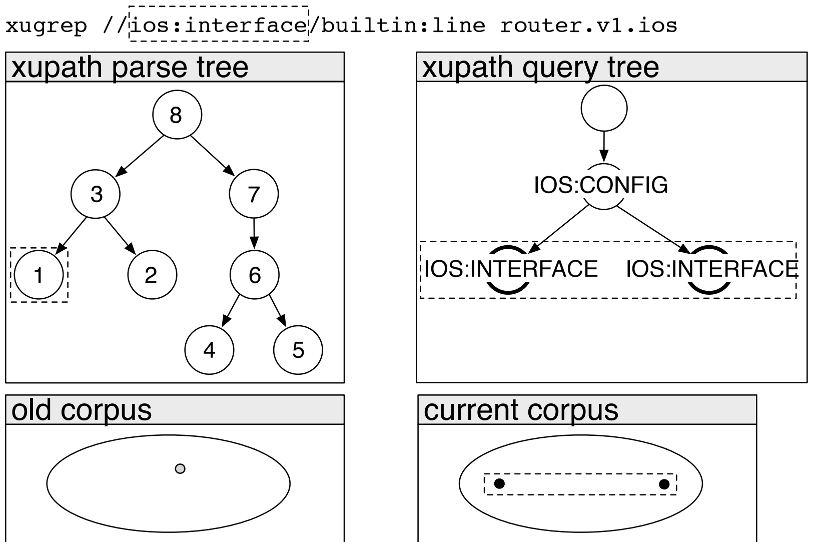


Figure 5.2: To extract all strings in the language of the xupath `xugrep` traverses the xupath parse tree in postorder. When `xugrep` arrives at a node for a production (`ios:interface`), `xugrep` extracts all strings in the language of that production from the current corpus. The results are used to instantiate a new corpus whose elements correspond to the leaf-level of the xupath query tree.

Figure 5.3: When `xugrep` visits another node in the xupath parse tree that corresponds to a production, it again extracts strings in the language of the production from the (old) corpus. In this example, we see how `xugrep` extracts lines from a corpus of interfaces. After the xupath parse tree has been completely traversed, `xugrep` outputs the contents of the current corpus.

relative to a subtree of our xupath query tree. Figure 5.4 illustrates how we may report matching lines relative to the interfaces and router configuration files in which they are contained.

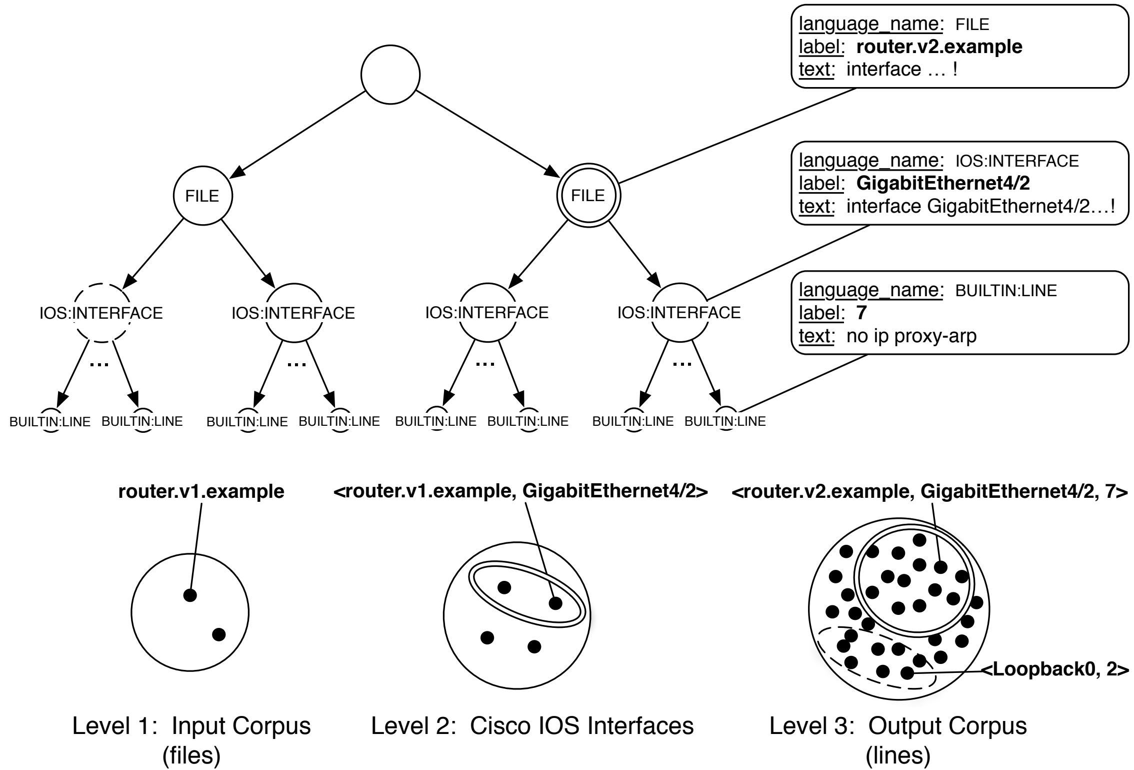


Figure 5.4: We may report matches that satisfy an xupath query in terms of surrounding high-level language structures. In this figure, we show that we can report matching lines relative to the interfaces and router configuration files in which they are contained.

Algorithm

We implement `xugrep`'s interpretation of xupath as follows. We use the input files to instantiate a result corpus that comprises a set of elements whose contents correspond to the file contents. `xugrep` outputs a *result corpus* that contains the corpus elements whose substrings satisfy the xupath query.

We then parse the xupath query. We perform a postorder traversal of the xupath parse tree. Since the vertices of XUTools parse trees are corpus elements, we check the language name of each vertex as we traverse.

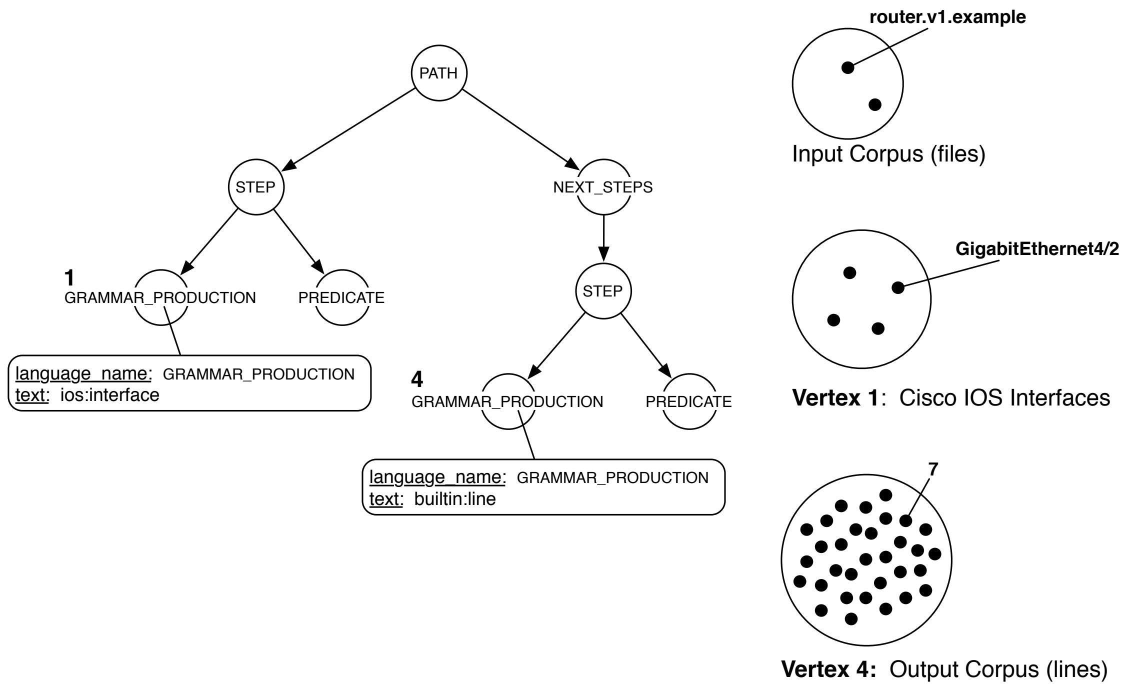
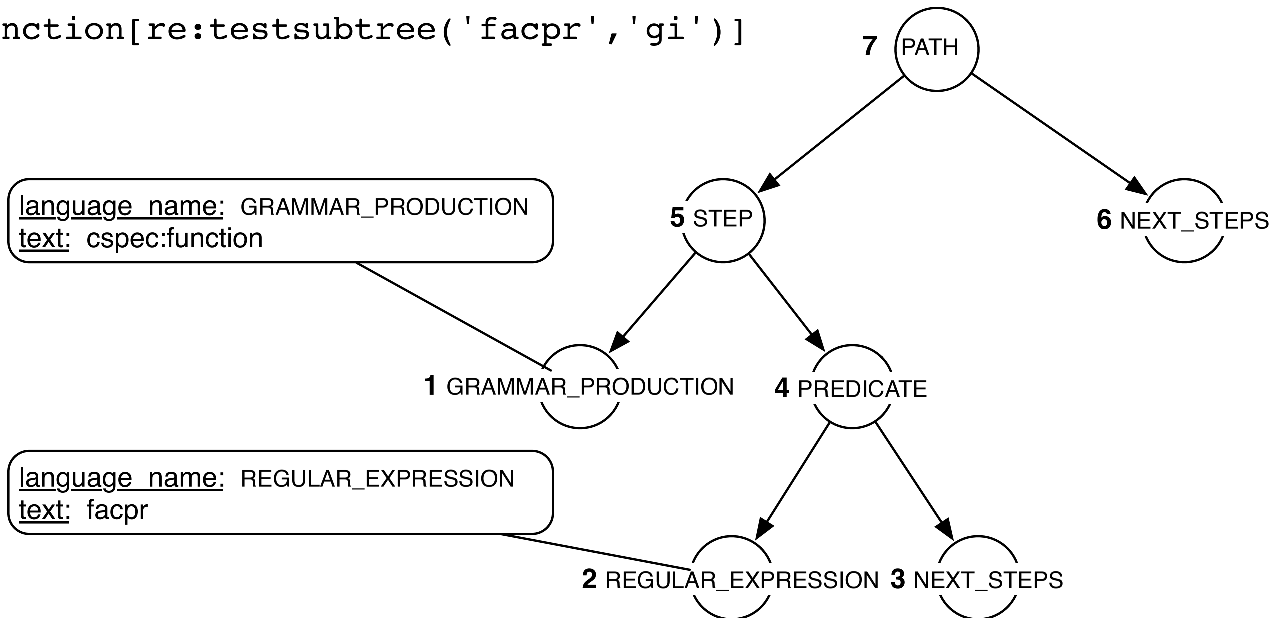


Figure 5.5: `xugrep` iteratively constructs a result corpus by a postorder traversal of the xupath parse tree. The first and second language names, correspond to the second and third levels of the xupath query tree shown in Figure 5.1.

If the value of the `language_name` field is GRAMMAR_ PRODUCTION, then we retrieve the string associated with that corpus element (the value of the `text` field. For example in Figure 5.5, at the first node, the grammar production is IOS:INTERFACE whereas it is BUILTIN:LINE at the fourth node (in postorder). When we encounter a node that represents a GRAMMAR_PRODUCTION, we use `scan_string` on every element in our result corpus to generate a new result corpus whose elements correspond to strings in the desired language (interfaces or lines in Figure 5.5).

Alternatively, if the language name is PREDICATE, then we can filter the result corpus for elements that satisfy the predicate condition.

After completing the postorder traversal of the xupath parse tree, we output the result corpus. Currently, we report the path from the result corpus element to the xupath query tree root.



position	result list	label-value paths	result language names
0	< "int\nputstr…return0;\n}" >	< <example.v1.c> >	<FILE>
1	< "int\nputstr(char *s)\n{ … }", "int\nfac(int n)\n{ … }", "int\nputn(int n)\n{ … }", "int\nfacpr(int n)\n{ … }", "int\nmain()\n{ … }" >	< <example.v1.c, putstr>, <example.v1.c, fac>, <example.v1.c, putn>, <example.v1.c, facpr>, <example.v1.c, main> >	< FILE, cspec:function>
2 - 7	< "int\nfacpr(int n)\n{ … }", "int\nmain()\n{ … }" >	< <example.v1.c, facpr>, <example.v1.c, main> >	< FILE, cspec:function>

Figure 5.6: This figure shows how `xugrep` constructs an report as it traverses the input xupath's parse tree. `xugrep` begins (at step 0) by placing the input files contents into a result list, the file names into the label-value paths list, and the FILE constant into the list of result language names. At step 1, `xugrep` uses the `parse` operation (introduced in Chapter 3) to extract all strings in the language of the production referenced by CSPEC:FUNCTION. The extracted strings form the new result list and the label-value paths are extended to include the `label` values for the new strings. Finally, at step 2, the result list is filtered so that only entries that contain the string *facpr* remain.

Implementation

We currently implement xugrep as a postorder traversal of the supplied xupath's parse tree. As we traverse this tree, we construct an xugrep report that contains (1) a result list of strings that satisfy the currently-processed xupath query, (2) a list of label-value paths for each of the strings, and (3) a list of language names that specifies the language associated with each element in a label-value path. Figure 5.6 illustrates how the xugrep generates the report as it walks the parse tree.

We implemented xugrep in Python using a functional programming style. We have one xugrep report update function for every type of xupath parse tree node: GRAMMAR_PRODUCTION, PATH, PREDICATE, REGULAR_EXPRESSION, and NEXT_STEPS. Our current implementation of xugrep is 191 lines.

5.2.2 XUWc Internals

In Chapter 4 we saw how one could use xuwc to count high-level language constructs and report counts relative to non-file contexts. In this section, we will explain how xuwc works.

Working Examples

In the following working examples, we explain in detail how xuwc works. In general, xuwc uses a xupath query tree to report counts. More information about the xupath query tree may be found in the previous subsection on xugrep.

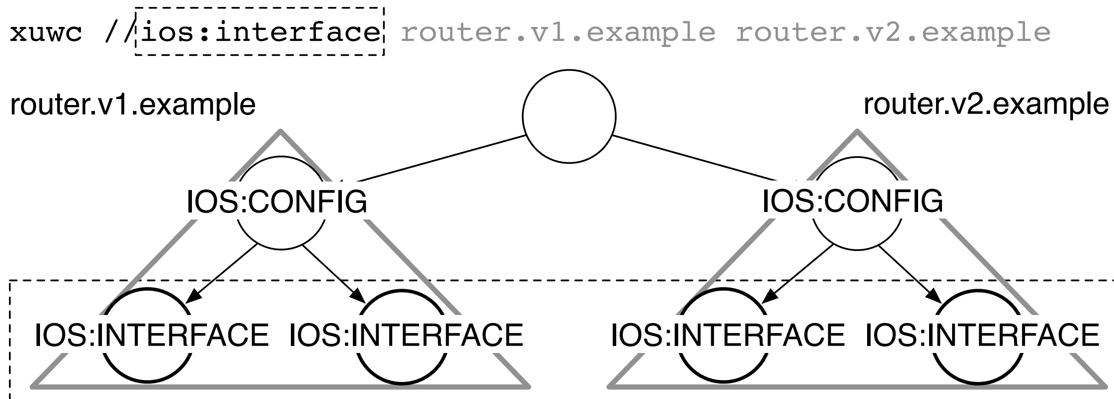


Figure 5.7: In the example above, we use `xuwc` to count the number of Cisco IOS interfaces within two configuration files. `xuwc` counts the number of interfaces (leaves in the xupath query tree) per file subtree (outlined in grey). Therefore, `xuwc` reports 2 interfaces in `router.v1.example` and 2 interfaces in `router.v2.example`.

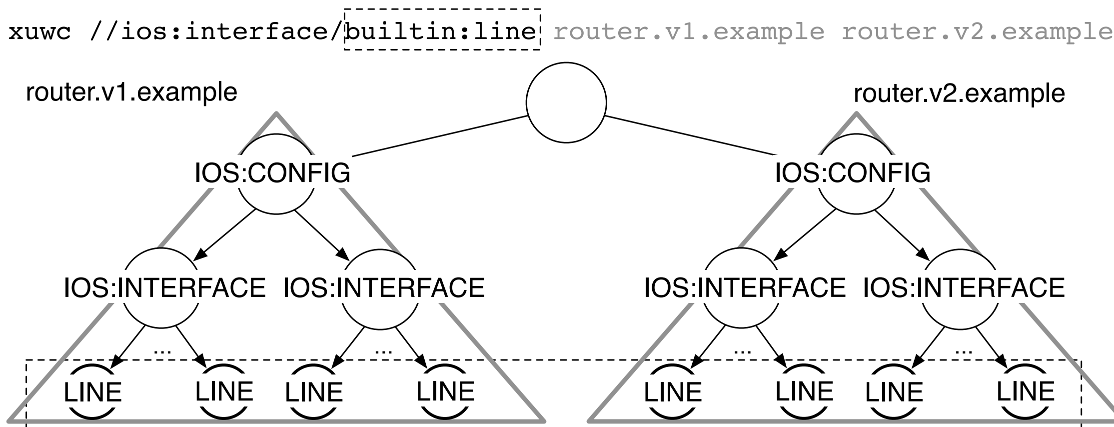


Figure 5.8: In this example, we can use `xuwc` to count the number of lines per Cisco IOS interface. Again, we see that `xuwc` will report the number of lines per file by counting the number of leaves per file subtree within the xupath query tree.

`xuwc` reports the number of high-level language constructs in the language of the xupath within each input file. Given a set of input files and a xupath, `xuwc` by default will count the number of leaves within each file subtree of the xupath query parse tree. Figure 5.7 illustrates this use case. When `xuwc` is invoked on two router configuration files with the xupath //IOS:INTERFACE, `xuwc` reports the number of interface leaves per file subtree. In this example, two interfaces are extracted from each of the two

router configuration files. Figure 5.8 shows how `xuwc` can report the total number of lines that belong to network interfaces using a similar procedure as in Figure 5.7.

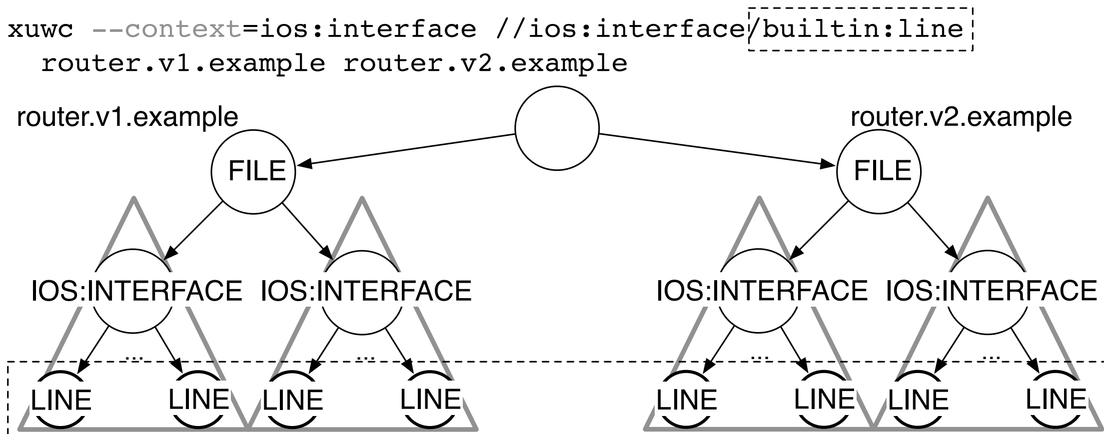


Figure 5.9: We can use `xuwc` to report counts relative to other high-level language constructs within a file. For example, by setting the `--context` flag to IOS:INTERFACE, `xuwc` will report the number of line leaves per interface subtree.

In Chapter 4, we mentioned that `xuwc` allows practitioners to report counts relative to non-file contexts via the `--context` flag. In Figure 5.9, `xuwc` once again constructs a xupath query tree, but reports the number of leaves per IOS:INTERFACE subtree rather than by FILE subtree (the default).

Algorithm

We currently implement `xuwc`'s interpretation of an xupath by processing the result corpus output by `xugrep`.

If the `--count` parameter is unspecified and the `--context` parameter is unspecified, then we return the number of results for each FILE subtree of the xupath query parse tree.

If the `--count` parameter is unspecified and the `--context` parameter is specified, then we partition the result set according to the subtrees rooted at the level of the context language, and return the number of results for each subtree.

If the `--count` parameter is specified and the `--context` parameter is unspecified, then we count the number of bytes, words, or characters per element in the result set and return the number of results for each FILE subtree of the xupath query parse tree.

If the `--count` parameter is specified and the `--context` parameter is specified, then we partition the result set according to the subtrees rooted at the level of the context language, count the number of strings in the language of the value of the `--count` parameter, and return the counts for each subtree.

In Chapter 6, we describe the implementation and evaluation of `xuwc`.

Implementation

We currently implement `xuwc` by processing an `xugrep` report for the input. First, we call `xugrep` on the xupath and input files to obtain an `xugrep` report.

We implemented `xuwc` in Python. The method that implements our algorithm in Chapter 5 is 96 lines. The total number of lines for the `xuwc` is 166 lines.

5.2.3 XUDiff Internals

Chapter 4 showed that `xudiff` compares two files in terms of high-level language structures that are specified by a context-free grammar. We will now describe *how* `xudiff` does this comparison.

Working Example

Figure 5.10 shows how `xudiff` works. Given two input files (in this case router configuration files), `xudiff` parses the contents of those files relative to the production in the xupath (IOS:CONFIG). Once the parse trees have been produced, `xudiff` uses the Zhang and Shasha tree edit distance algorithm to compute a matching between the two trees. Unmatched nodes in the parse tree of the first configuration file are

reported as deletions, while unmatched nodes in the second configuration files are insertions. Finally, updates may be reported when the labels of matched nodes differ. The example in Figure 5.10 shows that there are updates within the subtrees for the `Loopback0` interface blocks.

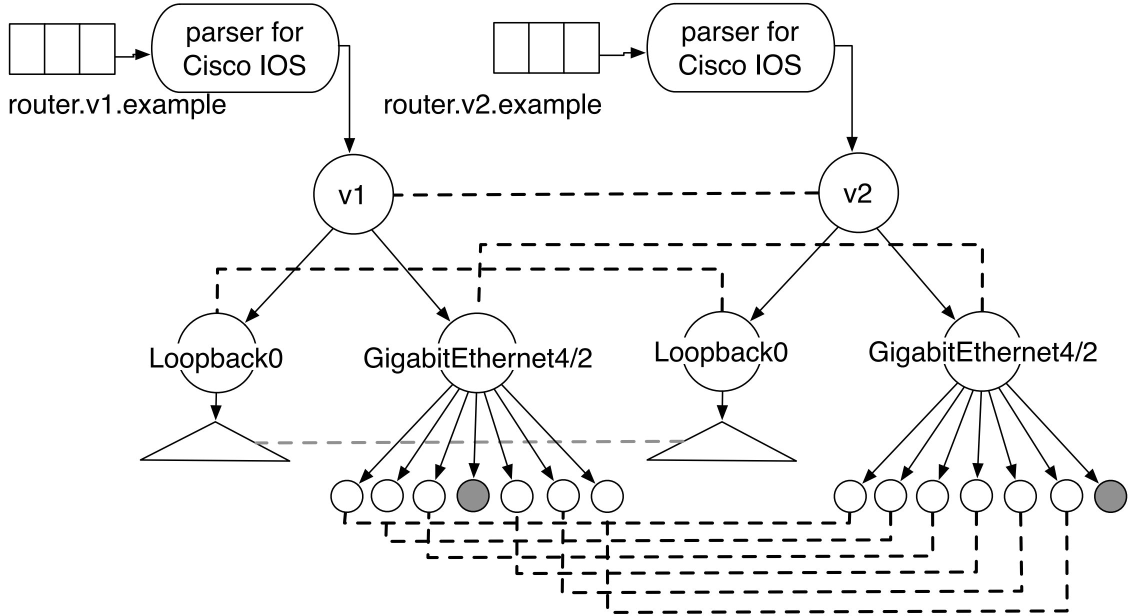


Figure 5.10: Practitioners may use `xudiff` to compare two network device configurations in terms of structures in a configuration language. In the example above, we see how `xudiff` compares the two router configuration files shown in Chapter 4. `xudiff` parses both input files relative to the production for Cisco IOS configuration files (IOS:CONFIG). `xudiff` then uses the Zhang Shasha Tree Edit Distance algorithm to compute a matching between the two trees. Unmatched nodes (in grey) in the first tree are deleted and unmatched nodes in the second parse tree are inserted. The grey dotted line between the subtrees for `Loopback0` indicates updated content within that subtree.

Algorithm

Our `xudiff` tool uses Zhang and Shasha's tree edit distance algorithm [151]. One nice property of this algorithm is that the tree edit distance is a metric, if the cost function is also a metric. More details about this algorithm may be found in the Zhang's dissertation [150].

Implementation

We have implemented the Zhang and Shasha tree edit distance algorithm [151]. This tree-edit distance is the basis for the `xudiff` tool.

We can currently compute edit scripts for parse trees in TEI-XML, Cisco IOS, and C. Our Python implementation of the algorithm is 254 lines.

5.2.4 Grammar Library

When a practitioner writes a regular expression or a context-free grammar, that practitioner specifies a computational machine to recognize a language. Although practitioners might not currently be able to write context-free grammars as quickly as they would write regular expressions, our strategy is to provide a library of grammars that satisfy a broad range of use cases.

We designed our grammar library to isolate references to language constructs from the encoding of those constructs much as an abstract data type separates a reference to an operation from that operation's implementation. We represent these constructs as a `language_name`. Practitioners already isolate language constructs from their encoding naturally: CAs reference *sections* and *subsections* of policy, and network administrators reference *interfaces*.[3] C developers, in order to use a library function in their own code, must know the name, purpose, and calling sequence of that function but not its implementation. Similarly, users of our grammar library need to know the name and purpose of the construct upon which they want to operate, but not its specification as a context-free grammar production, to process that construct via XUTools.

We designed XUTools to operate in terms of references to language constructs because the way in which people *reference* information remains relatively stable but the manner in which people encode information changes with technology. Consider

[3]Thanks to Dartmouth-Lake Sunapee Linux Users Group (DLSLUG) members for their feedback.

the historical transmission of text in which books and lines of Homer's Odyssey migrated from manuscripts, to books, to digital formats. Although the physical media and encoding of the text changed, the high-level constructs of book and line survived. In software engineering, the principle of an Abstract Data Type (ADT) echoes this philosophy—although an ADT's implementation may change over time, the interface to that implementation remains stable.

PyParsing Grammar Library

We currently implement a grammar library for XUTools with the PyParsing framework. More information about the grammar library may be found at the XUTools website `http://www.xutools.net/`.

- *XML Vocabularies*: We have currently implemented small grammars for NVD-XML and TEI-XML.
- *Cisco IOS*: We have implemented a grammar for a subset of Cisco IOS.
- *xupath*: We have implemented a grammar to parse xupaths based upon Zazueta's Micro XPath grammar [149].
- *C*: We are using McGuire's subset-C parser as a proof-of-concept for simple C source files [81].
- *builtin*: We have a `builtin` grammar for commonly-used, general-purpose constructs such as lines.

Table 5.1 illustrates the current sizes of grammars used by our XUTools. The table lists the grammar, the number of productions specified by the grammar, the number of productions in that grammar that we reference as language names, and the number of lines in the grammar. We note that the PyParsing API constructs a parser using a syntax that resembles a BNF grammar. As such, the number of lines also reflects the

number of lines necessary to implement the equivalent of a recursive-descent parser for the languages upon which each grammar is based.

grammar	number of productions	number of languages	number of lines
NVD-XML	6	3	8
TEI-XML	19	8	26
Cisco IOS	31	10	61
C	26	1	55
XPath	11	0	28
Builtin	1	1	2

Table 5.1: The sizes of the grammars used by our XUTools in terms of total number of productions, productions that we reference as language names with our XUTools, and number of lines to encode the grammar.

.

Since PyParsing productions specify functions that perform lexical and syntactic analysis, the productions listed above contain *token* definitions as well as grammar productions. More information about PyParsing may be found in Appendix B.

Parse Trees

Our XUTools were designed to operate on parse trees. Currently, we represent each vertex of a parse tree with a Python dictionary. Specifically, the dictionary for each parse tree node contains a key for the instance variables in a corpus element. In addition, each node dictionary has a `children` key that indexes a list of children that are ordered as specified by the right hand side of the nonterminal production.

5.2.5 xupath

Our goal for xupath was to implement a powerful and general-purpose querying syntax for structured texts, including but not limited to XML. Our intent is for practitioners to use this syntax to extract a corpus of language constructs that they want to

process. For example, in Chapter 3, Figure 3.11, we showed how to extract corpora of sections, subsections, and subsubsections from a parse tree for a PKI policy. We can reference these various corpora using the following xupaths: ///TEI:SECTION, //TEI:SUBSECTION, and //TEI:SUBSUBSECTION respectively. Later in this chapter, we will see more complicated examples of the xupath syntax.

Our xupath provides the necessary syntax for our XUTools to match strings in the language of a context-free grammar and to describe the context in which processing and reporting should occur. We observed that the XPath Query language [147] performs this function for XML documents. Therefore, we use an XPath-like syntax to express our queries on texts.

Practitioners want to process structured file formats besides XML, such as Cisco IOS, C, and JSON. Additionally, practitioners want to process formats that are *in between* traditional line-based formats and XML. In Figure 5.11 below, we see that configuration blocks—known as *directives*—may be nested arbitrarily and that blocks that correspond to modules may define their own configuration syntax. The contents of the `log_config_module` in Figure 5.11 has a specialized syntax to describe the format of logs. We can use xupath to query a corpus composed of varied formats because xupath syntax lets us compose references to constructs in a variety of languages.

Specification

An xupath query consists of a sequence of references to language constructs (each represented as a `language_name` in the libxutools grammar library). Consider the following three examples in which we describe the query and how the xupath syntax expresses that query.

First, we can express the set of Cisco IOS interface blocks relative to an input corpus via the xupath /IOS:INTERFACE. This xupath consists of a sequence of one language construct, the interface blocks as defined by Cisco IOS.

Apache server configuration example

```
<Directory "/Library/WebServer/Documents">
   Options Indexes FollowSymLinks MultiViews
   AllowOverride None
   Order allow, deny
   Allow from all
</Directory>

LogLevel warn

<IfModule log_config_module>
    LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined
    LogFormat "%v %h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combinedvhost
    LogFormat "%h %l %u %t \"%r\" %>s %b" common
    LogFormat "%v %h %l %u %t \"%r\" %>s %b" commonvhost

    <IfModule logio_module>
      # You need to enable mod_logio.c to use %I and %O
      LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" %I %O" combinedio
   </IfModule>
</IfModule>
```

Figure 5.11: The format of configuration files for the Apache server blends a XML-like syntax with a traditional line-based configuration. Configuration information for Apache server modules is contained in `IfModule` elements which resemble XML. Due to module dependencies, these configuration blocks may be nested. For example the `log_config_module` contains the `logio_module`. Furthermore, a module may specify its own configuration syntax. In this figure we see that the `log_config_module` has a syntax to specify a log format. We designed xupath to accommodate references to formats *in between* XML and traditional configuration formats.

Second, we can also use *predicates* to filter the set of Cisco IOS interface blocks to those that contain an access group security primitive.

//IOS:INTERFACE [re_testsubtree('access-group')].

Third, we can also use xupath to query a corpus in terms of constructs from several different languages. For example, we can express the set of lines contained within a C function as //CSPEC:FUNCTION/BUILTIN:LINE. Later in this chapter, we will describe in more detail the grammars and language constructs that are currently available in our grammar library.

Implementation

Since the xupath syntax is based upon XPath, we implemented xupath in Python as a modified MicroXPath [149] grammar ported to PyParsing [82]. Given an xupath query, our grammar generates a parse tree. Figure 5.12 shows an example parse tree. Our grammar generates a parse tree of six types of nodes shown in Table 5.2. We use the `language name` field to denote node type. xupath parse trees are implemented using Python dictionaries.

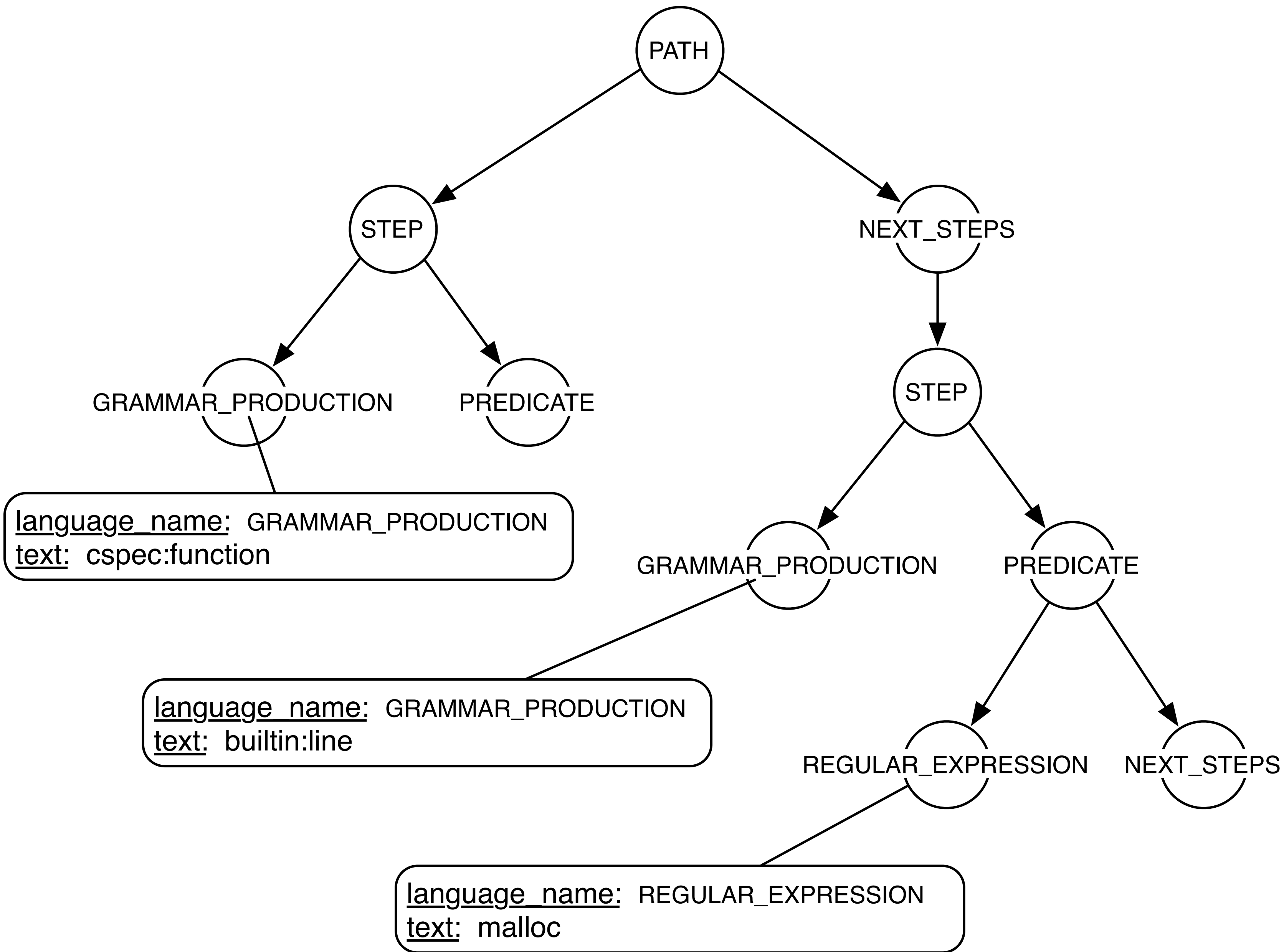


Figure 5.12: We implemented our xupath syntax as a modified MicroXPath [149] grammar. This grammar produces a parse tree with six types of nodes explained in Table 5.2.

Nodes in an XUPath Parse Tree		
node type	**"text" field value**	**description**
GRAMMAR_ PRODUCTION	the name of the language construct to apply to the current result set	A reference to a language construct. For example *ios:interface*, *tei:section*, and c*spec:function* are all possible values because they are language names that reference a grammar production.
PATH	n/a	the xupath parse tree root
PREDICATE	n/a	A regular expression paired with a list of steps to apply next. A predicate appears within square brackets in the XUPath (for example [re_testsubtree('malloc','gi')])
REGULAR_ EXPRESSION	a regular expression by which to filter the current result set.	A regular expression, for example '*malloc'*. When applied, the result set contains only strings that contain a substring that matches the regular expression.
STEP	n/a	A reference to a language construct paired with zero or more predicates. For example *ios:interface* is a language construct with no predicates while c*spec:function[re_testsubtree("malloc','gi')]* has a language construct and a predicate. XUPath steps are delimited by '/'.
NEXT_STEPS	n/a	A list of steps to apply to the result set. The list in left-to-right order according to how the steps appear in the XUPath.

Table 5.2: Our xupath grammar generates a parse tree of six types of nodes that we describe here. The examples in this table reference Figure 5.12.

5.3 Conclusions

In this chapter, we reinterpreted each of the three core limitations of security policy analysis as design requirements for text-processing tools and sketched how our XUTools modules meet those requirements. In the second section, we described the purpose, design, and behavior of each of our XUTools modules. In the next chapter (Chapter 6), we evaluate our XUTools in general. In Chapter 7, we will evaluate a specific application of XUTools that gives both network administrators and auditors new, practical capabilities to measure the security posture of their networks.

Chapter 6

General Evaluation of XUTools

In this chapter, we evaluate our XUTools. For each of our tools, we discuss any practitioner feedback or related work.

6.1 XUGrep

Our `xugrep` generalizes `grep`; `xugrep` extracts all strings in a set of files that match a xupath.

6.1.1 Evaluation—Qualitative

In Chapters 2 and 5 we motivated `xugrep` with several real-world use cases. We now briefly discuss how `xugrep` satisfied the use cases in Chapter 5.

Our first `xugrep` use case was inspired by practitioners at the RedHat Security Response Team. They wanted a way to parse and query XML feeds of the National Vulnerability Database (NVD-XML). During our discussion of xupath, we said that `xmllint` could satisfy this use case. However, our `xugrep` tool operates on a more general class of languages. We extended `xugrep` to operate on NVD-XML feeds so that one could extract the vulnerability scores of all NVD entries that contain the

string `cpe:/a:redhat`. Figure 6.1 shows some example output.

Our second `xugrep` use case allows practitioners to extract blocks of C code and is based upon discussions we had with practitioners at the LISA 2011 poster session as well as the subsequent discussions on Slashdot [107, 142].

6.1.2 Evaluation—Quantitative

We implemented `xugrep` as a postorder traversal of the xupath parse tree and so this takes linear time in the number of nodes in the xupath query. For the examples we have considered in this paper, an xupath query resolves to a handful of nodes. Nonetheless, when we visit a node whose production name is of type GRAMMAR_PRODUCTION or REGULAR_EXPRESSION, we must perform the `parse` operation on the current result corpus (introduced in Chapter 3). We iterate through each corpus element in our query result set and scan the element's `text` field for matches. We scan the element's `text` field with PyParsing's `scan_string` method.

As mentioned in Appendix B, the `scan_string` method takes $O(n^2)$ time and $O(n)$ space when packrat parsing is enabled (exponential time and linear space when disabled). At every step of the xupath we may apply the `scan_string` method at most twice. Therefore, given a xupath of k steps, where the largest corpus produced by `xugrep` has at most m corpus elements per step, and it takes $O(n^2)$ time to process each corpus element, then `xugrep` takes $O(kmn^2)$ time. However, we note that k (the number of steps in a xupath) is usually much less than n (the maximum number of characters in corpus element's `text` field). Therefore, `xugrep` may take $O(mn^2)$ time in the worst case. If at each step of the xupath, at each element of that step's corpus, PyParsing reuses space, then the `xugrep` should only use linear space in the size of the input string (measured in characters).

Our implementation of `xugrep` is 191 lines of Python. This is small enough to quickly port to other languages if desired. Furthermore, we have 438 lines of unit

xugrep NVD-XML example

```
bash$ xugrep -1 "//nvd:entry[re:testsubtree('cpe:/a:redhat','gi')]/nvd:score"
   nvdcve-2.0-2012.xml

FILE                  NVD:ENTRY       NVD:SCORE          TEXT
nvdcve-2_0-2012.xml   CVE-2012-0818   1                  <cvss:score>5.0</cvss:score>
nvdcve-2_0-2012.xml   CVE-2012-1106   1                  <cvss:score>1.9</cvss:score>
nvdcve-2_0-2012.xml   CVE-2012-1154   1                  <cvss:score>4.3</cvss:score>
nvdcve-2_0-2012.xml   CVE-2012-1167   1                  <cvss:score>4.6</cvss:score>
nvdcve-2_0-2012.xml   CVE-2012-2110   1                  <cvss:score>7.5</cvss:score>
nvdcve-2_0-2012.xml   CVE-2012-2333   1                  <cvss:score>6.8</cvss:score>
nvdcve-2_0-2012.xml   CVE-2012-2377   1                  <cvss:score>3.3</cvss:score>
nvdcve-2_0-2012.xml   CVE-2012-2662   1                  <cvss:score>4.3</cvss:score>
…
nvdcve-2_0-2012.xml   CVE-2012-4540   1                  <cvss:score>6.8</cvss:score>
```

Figure 6.1: We can use `xugrep` to extract the vulnerability scores of entries in the National Vulnerability Database that mention `cpe:/a:redhat`. This directly addresses a use case scenario that we received in email from the RedHat Security Response Team.

tests for this tool that validate behavior for example queries for subsets of TEI-XML, NVD-XML, Cisco IOS, and C.

6.1.3 Related Work

Industry

Currently, there are a variety of tools available to extract regions of text based upon their structure. The closest tool we have found to our design of `xugrep` is `sgrep` [68]. `sgrep` is suitable for querying structured document formats like mail, RTF, LaTeX, HTML, or SGML. Currently, an SGML/XML/HTML scanner is available but it does not produce a parse tree. Moreover, `sgrep` does not allow one to specify the context in which to report matches. Nonetheless, the querying model of `sgrep` is worth paying attention to.

If one is processing XML, XSLT [70] may be used to transform and extract information based upon the structure of the XML. We have already mentioned libxml2's `xmllint` [145] and its corresponding shell for traversing a document tree. Furthermore xmlstarlet has been around for a while and can also be used to search in XML files [60].

Cisco IOS provides several commands for extracting configuration files. For example, the `include` (and `exclude`) commands enable network administrators to find all lines in a configuration file that match (and don't match) a string. Cisco IOS also supports regular expressions and other mechanisms such as `begin` to get to the first interface in the configuration [29]. In contrast, our `xugrep` enables practitioners to extract matches in the context of arbitrary Cisco IOS constructs.

Windows Powershell has a `Where-Object` Cmdlet that allows queries on the properties of an object. An object may be created from a source file by casting it as a type (such as XML) [100].

Pike's structural regular expressions allow users to write a program to refine

matches based on successive applications of regular expressions [97]. Our approach is different because we extract matches based upon whether a string is in the language of the supplied grammar.

Augeas [80] is similar in spirit to our tools as it focuses on ways to configure Linux via an API that manipulates abstract syntax trees. This library includes a canonical tree representation, and path expressions for querying such trees. The goals of Augeas and XUTools are complimentary, Augeas provides an API to manipulate configuration files safely and XUTools extends existing text-processing tools so that practitioners can operate and analyze a variety of texts in terms of their high-level structures.

Academia

Although Grunschlag has built a context-free grep [59], this classroom tool only extracts matches with respect to individual lines. Coccinelle [31] is a semantic grep for C. In contrast, we want our tool to have a general architecture for several languages used by system administrators.

As noted in Chapters 2 and 5, our `xudiff` complements research in network configuration management. For example, Sun et al. argue that the block is the right level of abstraction for making sense of network configurations across multiple languages. Despite this, however, they only look at correlated changes in network configurations in Cisco [125]. Similarly, Plonka et al. look at stanzas in their work [98].

6.2 XUWc

As discussed in Chapter 5, our `xuwc` generalizes `wc` to count the number or size of strings in an xupath result set relative to some context.

6.2.1 Evaluation—Qualitative

In Chapter 2 and 5 we motivated `xuwc` with a variety of use cases involving Cisco IOS. In Chapter 7 we will see how `xuwc` satisfies these use cases so that practitioners can understand how network configurations evolve.

Additionally, `xuwc` may also be useful to look at code complexity over time. Just as we look at the evolution of network configuration complexity, we can start to look at the number of lines per function or even the number of functions per module over the entire lifetime of software.

6.2.2 Evaluation—Quantitative

We implemented `xuwc` as a routine to process the xupath query result set returned by `xugrep`. Therefore, the time-complexity of `xuwc` includes that of `xugrep`. If `xugrep` returns a result set of m elements, then `xuwc` must either count each of the elements in the result set exactly once or scan the `text` field of each of the elements in the result set exactly once. This depends upon whether the `--context` or `--count` flags have been specified. We consider the four cases of the `xuwc` algorithm (discussed in Chapter 5).

If neither `--count` nor `--context` flags have been specified, then `xuwc` counts the number of elements in the result set relative to the files from which the corpus element `text` fields were extracted. It takes constant time to check the `label-value path` stored in the corpus element (or `xugrep` report) to determine the file name. There are m corpus elements so `xuwc` takes $O(m)$ time in this case.

If the `--count` parameter is unspecified and the `--context` parameter is specified, then we must return the number of results for each context subtree. Again, it takes constant time to determine the appropriate subtree for a corpus element from its `label-value path`. There are m corpus elements so `xuwc` takes $O(m)$ time in this case as well.

If the `--count` parameter is specified, then regardless of the `--context` parameter, we must count the number of bytes, words, or characters per corpus element in the result set and return the number of results relative to the appropriate context. We've already seen that it takes constant time to determine the context subtree in which to report results, therefore we need to account for the time to count bytes, words, or characters per corpus element. In our evaluation of the `Parser` interface (in Appendix B, we explain that the `scan_string` method takes $O(n^2)$ with packrat parsing and $O(n)$ space with packrat parsing enabled for an input string of length n. Therefore, given m corpus elements and at most $O(n^2)$ language-construct extraction time per element, the worse-case time complexity is $O(mn^2)$ for `xuwc`. Again, if we assume that `scan_string` reuses space, then the space complexity for `xuwc` is $O(n)$.

Therefore, the worst-case complexity for `xuwc` is $O(mn^2)$ time and $O(n)$ space.

Our Python implementation of `xuwc` is currently 96 lines. We have 408 lines of unit tests for this tool that cover examples in TEI-XML, NVD-XML, Cisco IOS, and C.

6.2.3 Related Work

Academia

The general implication of `xuwc` is that it will allow practitioners to easily compute statistics about structures within a corpus of files in language-specific units of measure. We were unable to find prior work that attempts to generalize `wc`.

Industry

There are a number of word count tools used every day in word processors such as Word and emacs. These allow practitioners to count words, lines, and characters within a file.

There are also several programming-language utilities to compute source-code

metrics. The Metrics plugin for Eclipse allows one to count the number of packages, methods, lines of code, Java interfaces, and lines of code per method within a set of source files [84]. Vil provides metrics, visualization, and queries for C#, .NET, and VisualBasic.NET and can count methods per class and lines of code [135]. Code Analyzer is a GPL'd Java application for C, C++, Java, assembly, and HTML and it can calculate the ratio of comments to code, the lines of code, whitespace and even user-defined metrics [32].

Finally, Windows Powershell has a nice CmdLet called Measure-Object that allows people to gather statistics about an object [100].

6.3 XUDiff

The `xudiff` tool generalizes `diff` to compare two files (or the contents of two files) in terms of higher-level language constructs specified by productions in a context-free grammar.

6.3.1 Evaluation—Qualitative

Chapters 2 and 5 both motivate `xudiff` with RANCID changelogs as well as the need to understand changes to document-centric XML. We address the former use case when we show examples of the `xudiff` CLI in Chapter 5 as well as discuss how to measure the similarity of network security primitives in Chapter 7. We address the latter use cases when we talk about pilot studies in X.509 PKI and terms of service policies in Chapter 8.

We should note that one benefit of `xudiff` is the ability for practitioners to choose the right level of abstraction with which to summarize a change. For example, a developer could generate a high-level edit script by reporting changes in the context of functions or modules. In contrast, if an implementation of an API method changed,

then perhaps the developer would want to generate an edit script that describes changes in terms of lines within interfaces. We think that it would be interesting to measure to what extent we can reduce the size of an edit script by expressing changes in terms of different levels of abstraction. Understanding the size requirements for edit scripts at different levels of abstraction would have practical benefit for people that store change logs as evidence for audit. We leave exercise as potential future work.

6.3.2 Evaluation—Quantitative

Our `xudiff` design relies upon the Zhang and Shasha algorithm to compute edit scripts. The time complexity for this algorithm is
$O(|T_1| * |T_2| * \min(\mathit{depth}(T_1),\ \mathit{leaves}(T_1)) * \min(\mathit{depth}(T_2),\ \mathit{leaves}(T_2))\ )$ and its space complexity is $O(|T_1| * |T_2|)$ [151]. In these formulas, $|T_1|$ is the number of nodes in the first tree and $|T_2|$ is the number of nodes in the second tree.

Our Python implementation of Zhang and Shasha's algorithm is currently 254 lines. We have 563 lines of unit tests for this algorithm that tests every iteration of the example instance given in the Zhang and Shasha paper as well as an additional example based upon our TEI-XML dataset.

6.3.3 Related Work

Industry

The SmartDifferencer, produced by Semantic Designs, compares source code in a variety of programming languages in terms of edit operations based on language-specific constructs [38]. Unfortunately, the SmartDifferencer is proprietary. Finally, TkDiff [129], available for Windows, improves upon line-based units of comparison by highlighting character differences within a changed line.

Academia

The various components of our hierarchical diff tool use and improve upon the state-of-the-art in computer science. Computing changes between two trees is an instance of the tree diffing problem and has been studied by theoretical computer science [9]. Researchers have investigated algorithms such as subtree hashing, and even using XML IDs to align subtrees between two versions of a structured document and generate an edit script [23, 30]. Zhang and Shasha [151] provide a very simple algorithm for solving edit distance between trees that we currently use in `xudiff`.

Furthermore Tekli et al. in a comprehensive 2009 review of XML similarity note that a future research direction in the field woiuld be to explore similarity methods that compare "not only the skeletons of XML documents ... but also their information content" [127]. Other researchers have looked at techniques to compare CIM-XML for compliance [109], XML trees for version control [3], and Puppet network configuration files based upon their abstract syntax trees [134]. Recently, our poster that proposed XUTools [142] was cited in the context of differential forensic analysis [49].

6.4 Grammar Library

We designed our grammar library to isolate references to language constructs from the encoding of those constructs much as an abstract data type separates a reference to an operation from that operation's implementation. Our XUTools operate in terms of references to these language constructs because the way in which people *reference* information remains relatively stable but the manner in which people encode information changes with technology.

Anecdotal Feedback: We have presented our research at several different venues and practitioners have expressed concerns over the usability of our XUTools based

on how much knowledge a practitioner must have about data formats. For example, one reviewer at LISA 2012 stated that there is a debate in the system administration community as to whether XML should *ever* be read by a sysadmin. The reviewer went on to say that it is not easy to set up a search of a text when it requires a deep understanding of where things are in XML.[1]

Our grammar library interface directly addresses these concerns because practitioners do not have to understand *how* language constructs are encoded, but only *which* language constructs they want to use to analyze a text.

Some of our grammars such as Cisco IOS and TEI-XML were handwritten, while others, such as C, were adapted from extant work. We realize that the utility of our tools depends heavily upon the kinds of languages for which a xutools-friendly grammar exists. Therefore, our ongoing work on this problem considers two additional strategies beyond a handwritten grammar library based on feedback from practitioners and people in academia.

6.5 Conclusions

This chapter provided a general evaluation of our XUTools. Our qualitative evaluation demonstrated that our XUTools address the use cases discussed in Chapters 2 and 5. Where appropriate, this qualitative evaluation also included feedback from practitioners about our tools. Our quantitative feedback included the worst-case time and space complexity of our tools as well as test coverage. Finally, we evaluated our XUTools with respect to tools available in industry and academic research. In the next chapter, we will evaluate our XUTools against a specific use scenario: network configuration management.

[1] Conversations with reviewer 8D for our LISA 2012 paper.

Chapter 7

Application of XUTools to Network Configuration Management

We now evaluate our XUTools within the domain of network configuration management. We draw examples from enterprise networks as well as control networks in the electrical power grid. In Section 7.1, we briefly summarize the security policy analysis problems faced by network administrators and auditors.

In Section 7.2, we demonstrate that XUTools gives practitioners practical capabilities by enabling new computational experiments on structured texts.

Finally, Section 7.3 evaluates the novelty and utility of these capabilities by briefly reviewing related work and reporting anecdotal feedback that we received from real-world network administrators and auditors after we demonstrated these capabilities.

7.1 Introduction

Network administrators as well as network auditors, both in the enterprise and in power control networks, require the ability to summarize and measure changes to a

network. Network administrators must update configuration files in order to implement new services and to maintain security and compliance. If administrators don't update their configurations, then their networks may become less useful, vulnerable to attack, or non-compliant. Otherwise, if administrators do update their configurations, then they may introduce errors that lead to major network outages.

The work in this chapter directly addresses the three core limitations of security-policy analysis as they appear in the domains of network configuration management and power control networks. We discussed both of these problem domains in Chapter 2. In both domains, there is a need to be able to efficiently summarize and measure change to network configurations.

Our XUTools enable network administrators and auditors to answer practical questions about the evolving security posture of their network by addressing these limitations. Practitioners may use these tools to summarize and quantify changes to summarize and quantify changes to security primitives at various levels of abstraction within a network over time. This allows practitioners to see the big picture as well as to pinpoint specific changes that may cause bad behavior. Network administrators and auditors may use these same capabilities to gather evidence for other kinds of changes during network compliance audit.

In this chapter, we use XUTools to measure the evolution of access-control lists and roles within the Dartmouth College network from 2005-2009.[1] We note that although we focus on Access Control Lists (ACLs) and roles (object groups in Cisco IOS), we could just as easily apply similar analyses to other high-level language constructs.

[1]We obtained this dataset from Paul Schmidt at Dartmouth Computing Services. The years 2005-2009 were the most convenient for him to give me because they were all in the same version-control system.

7.2 XUTools Capabilities for Network Configuration Management

Our XUTools enables new computational experiments on structured texts, the results of which give practitioners capabilities that address our three core limitations of security policy analysis. For example, our XUTools-based capabilities can answer the following questions which are of practical importance to network administrators and auditors in the field.

- What are the object groups and ACLs in a set of network devices and how many of each are there?
- How similar are the object groups and ACLs in a set of network devices?
- Which ACLs are actually used in a network and where are they used?
- How do the answers to the previous three questions change over time?

The answer to each of the preceding questions is important for network administrators and auditors alike. We discuss each question in its own section. For each question we will describe its importance, our experimental set-up to answer the question, the results of our experiment, and our interpretation of the results.

7.2.1 Inventory of Network Security Primitives

Network administrators and auditors both need to understand which object groups and access-control lists are configured on a network.

The terminology Cisco chose for their access-control security primitives overlaps with terminology from traditional access-control literature. Therefore, we will briefly relate Cisco IOS terminology to the traditional access-control terminology found in the literature.

Access-control rules are traditionally expressed as a matrix whose rows are subjects and whose columns are objects. As stated by Smith and Marchesini, a subject S corresponds to an entity that performs an action and may be a user or a program. An object O corresponds to an entity that is acted upon and may be a directory or file. An entry in this matrix (S, O), contains rights that subject S may perform upon object O. Example rights include the ability to read or write to a file object [120]. Traditionally, an access-control list represents the information in the access-control matrix in column-major order—each object has a list of which subjects may act upon it [34].

For our purposes, a Cisco IOS access-control list specifies which devices, IP addresses, and protocols may access a network interface. A network interface is the point of connection between a network device and other parts of the network. A Cisco IOS ACL is applied to a network interface (an object in traditional terminology) and lists which devices, IP addresses, and protocols (subjects in traditional terminology) may act upon it.

In very large networks however, the number of entries in an ACL may reach hundreds of lines [34]. Furthermore, entries in an ACL may change and this puts a burden on network administrators. In order to reduce this burden and simplify ACL configuration, Cisco IOS introduced object groups that allow administrators to classify users, devices, or protocols into groups. These groups may then be referenced to define an ACL.

The Cisco IOS terminology can be confusing because Cisco IOS object groups specify a set of entities that are *subjects* or *objects* under traditional access-control terminology. When viewed as subjects, Cisco IOS object groups are analogous to roles in Role-Based Access Control (RBAC). For example, an administrator may use an object group to restrict a group of user machines (Cisco IOS object group as traditional access-control subject) to a set of applications that run on a specific

port on a specific set of servers (Cisco IOS object group as traditional access-contol object).

Network security evolves because organizations and the environments in which they are based change. Therefore, it is useful for an administrator or an auditor to be able to extract the object groups and ACLs that are defined on a network at a given point in time. A version-control repository combined with our XUTools gives practitioners this capability.

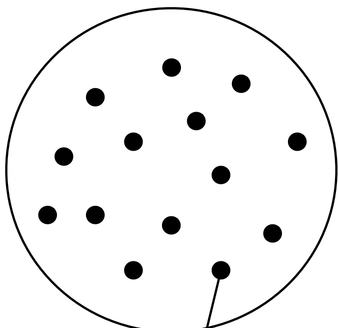
Set-up

Figure 7.1 illustrates how we implement this capability. Given a set of network devices, such as the set of routers and switches in the *core* or *wireless* networks, we define two corpora: the set of object groups and the set of access-control lists. We calculate the size of each object group and access control list in terms of the number of lines. Our `xuwc` allows us to perform this calculation because it enables us to count the number of lines in an occurrence of a language construct such as an object group or an ACL. In contrast, traditional `wc` lets practitoners count lines but only in the context of the entire input file. (We note that alternatively, we could define size by counting a construct besides line and we discuss alternative measures of size later in Section 7.3.) Finally, we report the total number of elements in each corpus, statistics about the size of each element (minimum, average, and maximum), and the top 10 largest elements.

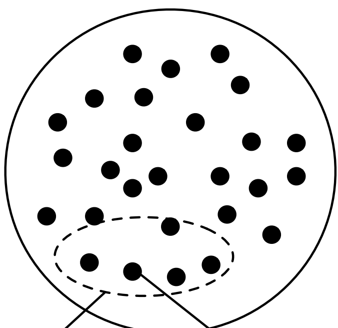
Results and Interpretation

We performed the above analysis on the latest versions of the *border*, *core*, *edge*, *voip*, and *wireless* networks. Table 7.1 summarizes our results below.

Input Corpus
(Dartmouth Core Devices)



Step 1: Extract Roles



Step 2: Sort Roles by Size

device	role	size (lines)
outside.switch...	phone_access	21
outside.etna1...	juniper_vpn...	14
⋮		
berry1-crt...	vf_amino_...	2

Figure 7.1: Given an input corpus of configuration files, we extract security primitives such as role, compute the *size* of each role, and report our results.

Object Groups in the Dartmouth Network Circa 2009		
network	**number**	**size (min/avg/max)**
border	0	0/0/0
core	124	2/4.0/21
edge	0	0/0/0
voip	0	0/0/0
wireless	0	0/0/0

ACLs in the Dartmouth Network Circa 2009		
network	**number**	**size (min/avg/max)**
border	18	2/31.0/80
core	64	2/7.0/39
edge	8	2/5.0/10
voip	1506	2/5.0/10
wireless	0	0/0/0

Table 7.1: We used our XUTools library to inventory the object groups and access-control lists across the Dartmouth network. We can see that all of the Object Groups are defined in the *core* network, while the largest ACLs are in the *border*.

Our results pinpoint which object groups and ACLs are the largest. For example, we know that the object groups are exclusively defined in the *core* network. In contrast, ACLs are defined throughout the network with the most being in the *voip* network, but the largest being in the *border* network. We can then use this information to look at the largest ACLs and see why they are so large.

7.2.2 Similarity of Network Security Primitives

Once we have the number and size of object groups and access-control lists on different parts of the network, we can start to compare them. Object groups group users, devices, or protocols. ACLs filter traffic into and out of an interface or VLAN and do so by referencing object groups.

Although Cisco designed object group-based ACLs to simplify management they can still grow large. Furthermore, object groups themselves may be defined or re-defined over time. Object groups are similar to roles in Role-Based Access Control (RBAC) and may drift or be copied and renamed. (In our lab's fieldwork with a very

large banking corporation, we found that just managing the roles can be an impractical problem and coming up with the right set of roles is infeasible [114].) Therefore, we employ clustering in order to try to make sense of which object groups and ACLs are most similar to one another. Again, the XUTools library gives us this capability.

Set-up

We employed two methods to investigate the similarity of ACLs and object groups: a size/name-based approach based on the results from Experiment 1 and a graph-based clustering approach.

First Approach: In the first approach, we compared ACLs from Experiment 1 whose sizes and names were roughly similar. To perform this comparison, we extracted and compared the appropriate ACLs using our XUTools. Figures 7.2 and 7.3 illustrate this process. The first figure shows two ACLs that have similar names and line counts (`Pharmacy-POS-out` and `Pharmacy-POS-in` respectively). The second figure shows the result of comparing these two ACLs using `xudiff` with different cost functions.

device	ACL	size (lines)
berry1-crt	VoIP_Call_Managers	39
⋮		
ropeferry1-crt	Pharmacy-POS-out	10
ropeferry1-crt	Pharmacy-POS-in	10
⋮		
berry1-crt...	vf_amino_...	2

Figure 7.2: We used the inventories of ACLs from our first experiment to find similarly-named ACLs that we can compare. In this example, we see that `Pharmacy-POS-out` and `Pharmacy-POS-in` are similarly named and the same number of lines long.

xudiff comparisons

```
bash-ios-ex1$ xudiff "//ios:accessList" ropeferry1-crt.dartmouth.edu.Pharmacy-POS-in
  ropeferry1-crt.dartmouth.edu.Pharmacy-POS-out

OPERATION      TOTAL COST     COST       MAPPING
Update         7.0            1.0        Pharmacy-POS-in (IOS:ACCESS_LIST) -> Pharmacy-POS-out (IOS:ACCESS_LIST)
…
Update         3.0            1.0        Pharmacy-POS-in, permit udp host 10.64.104.5 host 127.170.17.4 … (BUILTIN:LINE) ->
                                          Pharmacy-POS-out, permit udp host 129.170.17.4 host 10.64.104.5 … (BUILTIN:LINE)
Update         2.0            1.0        Pharmacy-POS-in, permit udp host 10.64.104.5 host 127.170.16.4 … (BUILTIN:LINE) ->
                                          Pharmacy-POS-out, permit udp host 129.170.16.4 host 10.64.104.5 … (BUILTIN:LINE)
Update         1.0            1.0        Pharmacy-POS-in, remark dns for pharmacy-pos (BUILTIN:LINE) ->
                                          Pharmacy-POS-out, remark dns response for pharmacy-pos (BUILTIN:LINE)


bash-ios-ex2$  xudiff --cost=text_word_edist "//ios:accessList" ropeferry1-crt.dartmouth.edu.Pharmacy-POS-in
  ropeferry1-crt.dartmouth.edu.Pharmacy-POS-out

OPERATION      TOTAL COST     COST       MAPPING
Update         18.0           1.0        Pharmacy-POS-in (IOS:ACCESS_LIST) -> Pharmacy-POS-out (IOS:ACCESS_LIST)
…
Update         5.0            2.0        Pharmacy-POS-in, permit udp host 10.64.104.5 host 127.170.17.4 … (BUILTIN:LINE) ->
                                          Pharmacy-POS-out, permit udp host 129.170.17.4 host 10.64.104.5 … (BUILTIN:LINE)
Update         3.0            2.0        Pharmacy-POS-in, permit udp host 10.64.104.5 host 127.170.16.4 … (BUILTIN:LINE) ->
                                          Pharmacy-POS-out, permit udp host 129.170.16.4 host 10.64.104.5 … (BUILTIN:LINE)
Update         1.0            1.0        Pharmacy-POS-in, remark dns for pharmacy-pos (BUILTIN:LINE) ->
                                          Pharmacy-POS-out, remark dns response for pharmacy-pos (BUILTIN:LINE)
```

Figure 7.3: After identifying similarly-named ACLs, we can compare them using our `xudiff`. When we viewed the edit script using a word-based cost function, we noticed that the IP addresses were flipped and slightly modified in the `permit udp` lines. This makes sense given that one ACL looks to permit outbound UDP traffic, while the other permits inbound UDP traffic.

Second Approach: In the second approach, we clustered object groups and ACLs based on the editing distance between their parse trees. Given a set of routers (the core network for example), we constructed the corpus of Object Groups and the corpus of ACLs as in the first experiment.

We processed each corpus to construct a *similarity graph* ($G = (V, E)$) where our vertices (V) are a set of corpus elements such as object groups or ACLs. The set of edges (E) corresponds to pairs of corpus elements and we weight these edges by a similarity function. Our similarity function s maps a pair of corpus elements (x_i and x_j) to the interval $[0, 1] \in \mathbb{R}$. The closer $s(x_i, x_j)$ is to 1, then the more similar the corpus elements are to one another. In order to see which pairs of corpus elements are the most similar, we remove all edges that have weight less than a similarity threshold [136].

For example, we can construct a similarity graph for the set of roles in a router configuration using the corpus of roles as our set of data points and a normalized tree edit distance as our similarity function. Specifically, we defined the normalized tree edit-distance between two data points x_i and x_j to be the minimum number of nodes required to transform the parse tree for x_i into the parse tree for x_j where inserts, deletions, and updates all have cost 1 (unit cost metric). We normalize this distance by the maximum number of nodes in the parse trees for x_i and x_j.

In this experiment, we connected all points that are at least 60% similar (in terms of parse tree nodes) but not identical. We note that filtering edges by a threshold allows us to "dial-in" the degree of similarity that we want between parse trees.

Results and Interpretation

We performed the above analysis on both object groups and ACLs. We focus on the ACLs in our first approach and object groups in our second. This is because object groups are not named similarly and so we actually cannot employ a size/name-based

approach to compare object groups.

First Approach: Since the ACLs in the *border* and *core* networks sometimes have similar names, we used our inventory of ACLs from Experiment 1 to determine which ACLs to compare. We compared the ACLs in the *border* network that were the largest and therefore, we hypothesized, more complicated to maintain. For the *core* network, we compared ACLs with similar names. Table 7.2 shows our results.

Using our `xudiff`, we can determine exactly where two similarly-named ACLs differ (if at all). The identically-named large *border* ACLs on `border1-rt` and `border2-rt` were identical. We detected differences, however, in the ACLs on the *core* routers. For example on `etna1-crt`, the 23-line ACLs `to_SN3407` and `to_SN3407_2` differed by a remark and `permit ip` command. The 10-line `Pharmacy-POS-out` and `Pharmacy-POS-in` differed by 7 lines, but by only 18 words. Upon closer inspection, we noted that IPs were flipped and slightly modified in the `permit` commands as expected for in and out filters.

Second Approach: As mentioned before, a name-based approach to compare Object Groups did not work because the names of the Object Groups we inventoried in Experiment 1 were quite different. Therefore, we clustered Object Groups by a measure based on tree editing distance. Figure 7.4 shows our results. We note that after trying a variety of percentages, the graph that had an edge for parse trees at least 60% similar was the one that balanced meaningful similarity with good-sized clusters.

ACLs in the Dartmouth Network Circa 2009				
	router/ACL 1 (size)	**router/ACL2 (size)**	**distance**	**comments**
border	border1-rt/from_l1_filter (76)	border2-rt/from_l1_filter (76)	0 lines	identical
	border1-rt/to_campus_filter2 (48)	border2-rt/to_campus_filter2 (48)	0 lines	identical
	border1-rt/to_border_filter (42)	border2-rt/to_border_filter (42)	0 lines	identical
core	etna1-crt/to_SN3407_2 (23)	etna1-crt/to_SN3407 (23)	3 lines	name update, remark update, new 'permit ip'
	ropeferry1-crt/Pharmacy-POS-out (10)	ropeferry1-crt/Pharmacy-POS-in (10)	7 lines	diff too coarse
	ropeferry1-crt/Pharmacy-POS-out (10)	ropeferry1-crt/Pharmacy-POS-in (10)	18 words	IP addresses 'flipped' on permits
	berry1-crt/DL150_out (6)	berry1-crt/CL150_out2 (7)	1 line	new 'permit ip'
	berry1-crt/DL150_out2 (7)	switchroom1-crt/CL150_out2 (7)	0 lines	identical

Table 7.2: We compared ACLs in the *border* and *core* networks based upon name and size-similarity via our ACL inventory from Experiment 1. We observe that versions of the large *border* ACLs were identical. We also note that in comparing the *Pharmacy* ACLs in *core*, we detected a “flipping” of IP addresses.

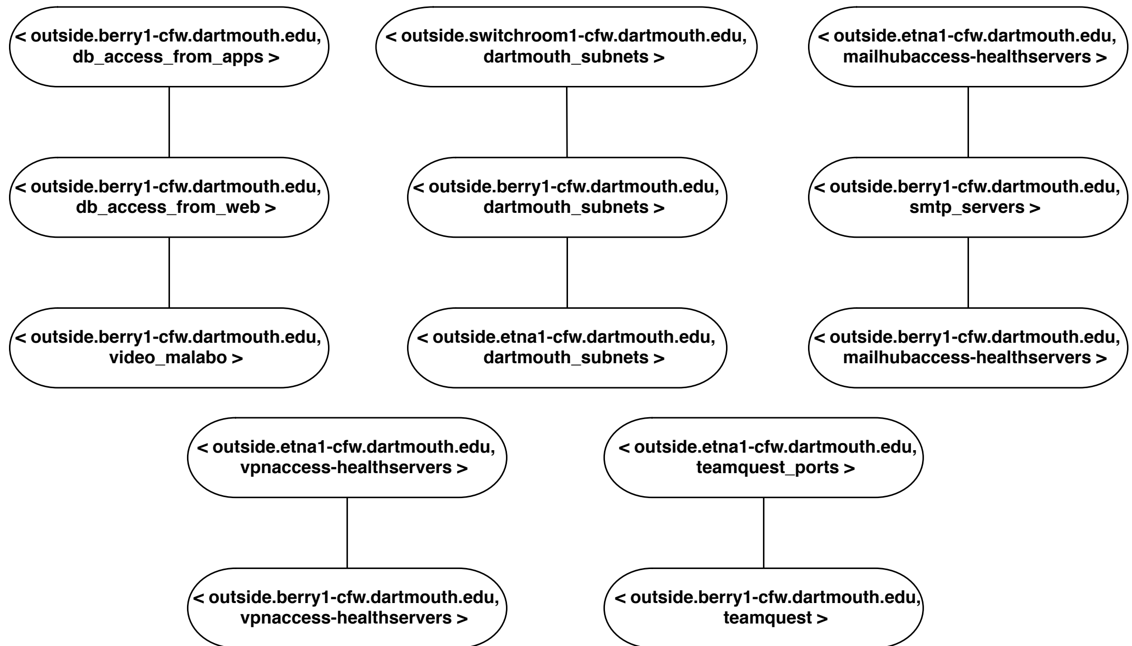


Figure 7.4: We clustered Object Groups by tree edit distance within the Dartmouth 2009 core network. Out of 124 Object Groups, we obtained 113 clusters. Edges between a pair of vertices indicate that the corresponding Object Groups are at least 60% similar. This figure shows a few of the clusters.

The edges in Figure 7.4 identify pairs of object groups that are only slightly different (and not identical). In Figure 7.5 we see that the two, very differently named Object Groups `mailhubaccess-healthservers` and `smtp_servers` are quite similar in their content. Figure 7.6 shows `xudiff` commands that correspond to the edges in this graph and highlights the ability to use our tools to *drill down* from a big-picture view of the network to lower-level details.

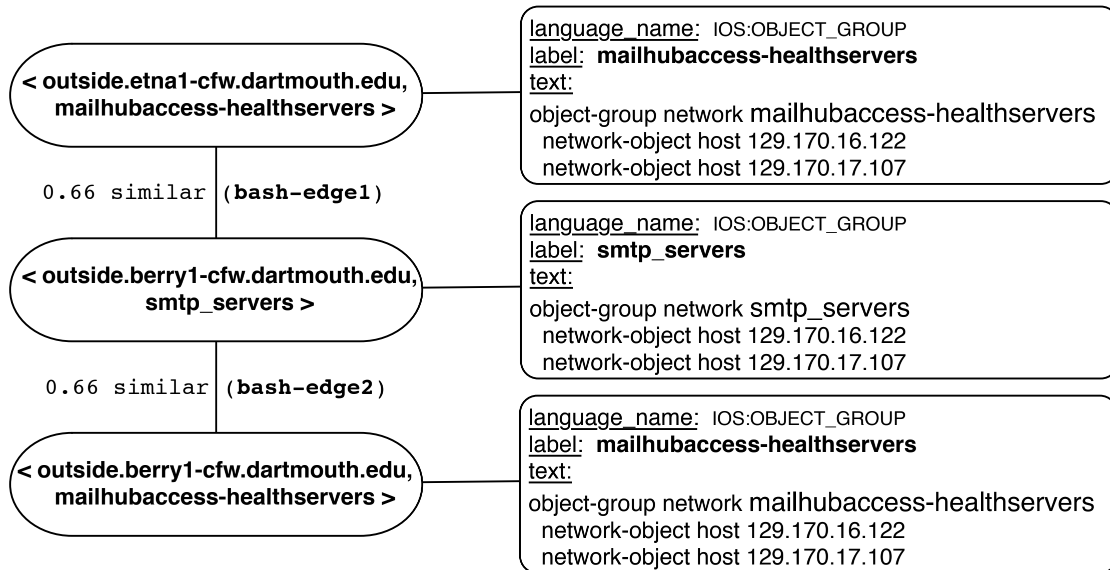


Figure 7.5: This sample cluster, taken from Figure 7.4 highlights that two roles, although named differently, may still have similar content. This shows that we can use our XUTools to help practitioners measure the similarity of roles within a network.

In addition, other edges yielded interesting but small changes. For example, consider the object groups named `vpnaccess-healthservers` on the routers `outside.-etna1` and `outside.berry1`. This pair showed up with a similarity threshold of at least 80% similarity. There was a 1 line change in the 4 line object group in which a network-object reference was changed from `juniper_vpn_tech_services` to `juniper_vpn_tech_services_etoken` and an IP address changed. The pairing `teamquest_ports` and `teamquest` on `outside.etna1` and `outside.berry1` had 1 line different, a single `port object`. This pair showed up with a similarity threshold of at least 70%.

7.2.3 Usage of Network Security Primitives

Although Experiment 1 demonstrated how we can use XUTools to inventory the defined network-security primitives, practitioners also find it helpful to know which primitives are used and where.

xudiff comparisons

```
bash-ios-ex1$ xudiff "//ios:objectGroup" outside.etna1-cfw.dartmouth.edu.mailhubaccess-healthservers
  outside.berry1-cfw.dartmouth.edu.smtp_servers

OPERATION      TOTAL COST     COST       MAPPING
Update         1.0            1.0        mailhubaccess-healthservers (IOS:OBJECT_GROUP) -> smtp_servers (IOS:OBJECT_GROUP)
Update         0.0            0.0        mailhubaccess-healthservers, network-object … (BUILTIN:LINE) ->
                                          smtp_servers, network-object … (BUILTIN:LINE)
Update         0.0            0.0        mailhubaccess-healthservers, network-object … (BUILTIN:LINE) ->
                                          smtp_servers, network-object … (BUILTIN:LINE)

bash-ios-ex2$ xudiff "//ios:objectGroup" outside.berry1-cfw.dartmouth.edu.smtp_servers
  outside.berry1-cfw.dartmouth.edu.mailhubaccess-healthservers

OPERATION      TOTAL COST     COST       MAPPING
Update         1.0            1.0        smtp_servers (IOS:OBJECT_GROUP) -> mailhubaccess-healthservers (IOS:OBJECT_GROUP)
Update         0.0            0.0        smtp_servers, network-object … (BUILTIN:LINE) ->
                                          mailhubaccess-healthservers, network-object … (BUILTIN:LINE)
Update         0.0            0.0        smtp_servers, network-object … (BUILTIN:LINE) ->
                                          mailhubaccess-healthservers, network-object … (BUILTIN:LINE)
```

Figure 7.6: This figure shows the ability to *drill down* within the similarity graph shown in Figure 7.5 and see how actual corpus elements differ via our `xudiff`.

Set-up

We used `xuwc` and `xugrep` to understand how many and which ACLs from Dartmouth's core network are applied to all `interface` and `class-map` commands. Specifically, we looked at usage of ACLs in the Dartmouth core network as of June 1, 2009.

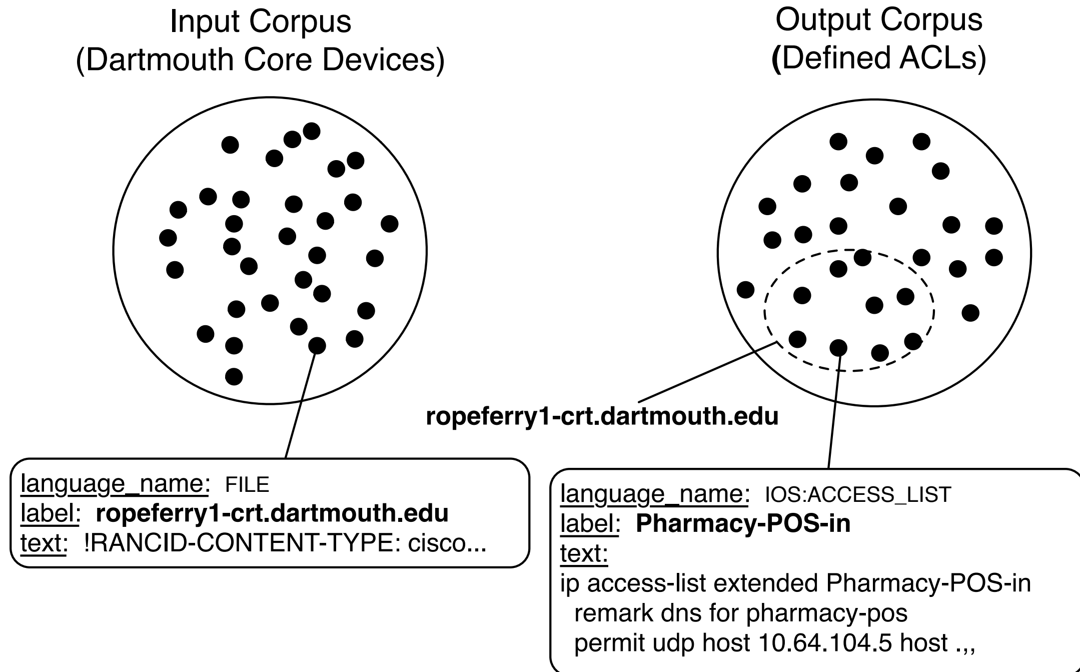


Figure 7.7: We used our `xuwc` to determine the number of access lists that were defined in the Dartmouth core network devices as of June 1, 2009. `xuwc` takes the configurations of Dartmouth core network devices (`core.2009/configs/*.edu`) as input. This input, shown as a set of corpus elements on the left, is processed by `xuwc` to count the number of ACLs (IOS:ACCESSLIST) within each router. The dotted line in the output set shows those ACLs that were defined on the `ropeferry1-crt.dartmouth.edu` router. The boxes pointing to individual corpus elements, show the values of the `CorpusElement` fields.

Figure 7.7 illustrates how we used `xuwc` (or `xugrep`) to compute how many (or which) ACLs were defined in the Dartmouth Core Network as of the summer of 2009.

Figure 7.8 shows how we used `xugrep` to compute how many (and which) *unique* ACLs are applied to all `interface` commands.[2] We used `xugrep` to extract the `ip`

[2] Please see Section 7.3 for a discussion on why we could not use `xuwc`.

access-group commands contained within each interface over all network devices in *core*. This was accomplished by invoking xugrep with a xupath of //IOS:INTERFACE/-IOS:ACCESSGROUP. This argument tells xugrep to extract all access group commands within any interface blocks. We then piped this output into sort and uniq to determined the unique ACL names. We performed a similar analysis to determine how many unique ACLs were used within Cisco IOS class-maps.

Results and Interpretation

We found that there are 21 ACLs applied to interfaces in the core ip access-group commands. Compare this to Experiment 1, in which there were 64 ACLs in total. By invoking the same pipeline with an xupath of //IOS:CLASSMAP/IOS:ACCESSGROUP, we found that there is one ACL applied to class-maps in the core.

These results are interesting to network administrators and auditors because they now have a way to understand how many of the ACLs that were *defined* in the Dartmouth core network are actually *used*. This is an important capability because administrators or auditors can focus in on those ACLs when debugging the behavior of the network's access-control policy.

Furthermore, this capability may give administrators the ability to understand which ACLs may be removed from a policy because they are no longer in use. This can help reduce network complexity that occurs when administrators are afraid to remove configurations that might be important. As mentioned before, Benson et al. note that complexity of a network increases with maintenance and changes to configurations [8]. These results let administrators know that only 21 out of 64 defined ACLs are actually applied to a network interface and our results pinpoint how access-control policy implementations could be simplified by removing unused ACLs.

```
bash-ios-ex2$ xugrep -RE=LE "//ios:interface/ios:accessGroup" core.2009/configs/*.edu
  | cut -f 3 | sort | uniq -c | sort
```

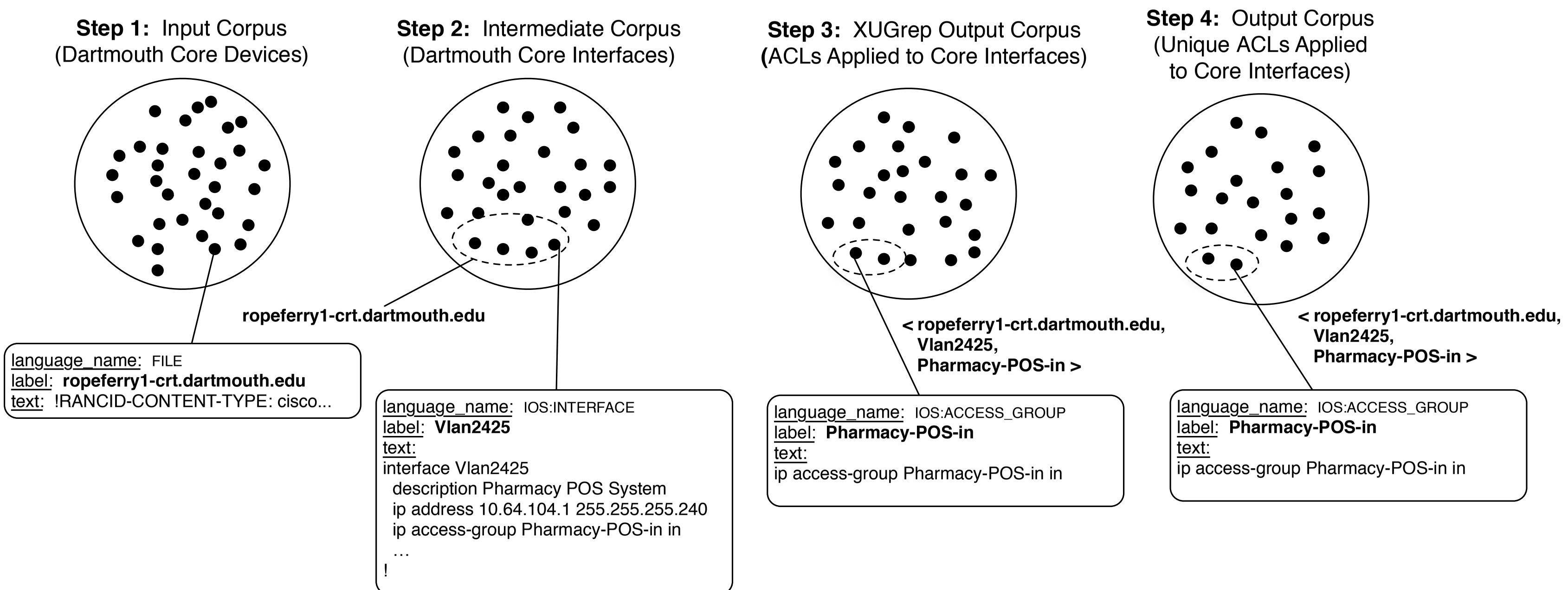


Figure 7.8: We used `xugrep` to determine the *unique* number of access lists that were applied to a network interface in the Dartmouth Core network as of June 1, 2009. Given a set of input files `core.2009/configs/*.edu`, `xugrep` instantiates an input corpus and first extracts a set of interfaces (IOS:INTERFACE). `xugrep` then extracts occurrences of the `access-group` command (IOS:ACCESSGROUP) from the set of interfaces. Access groups apply ACLs to interfaces. `xugrep` outputs one line per access group (via the `--R2=LE` option) we redirect this output into a Unix pipeline to calculate the unique ACLs that were applied to the input routers' interfaces and sort them lexicographically.

7.2.4 Evolution of Network Security Primitives

Network administrators and auditors alike both need to understand how the security of a network evolves. In the following experiment, we apply our capabilities from the first three experiments to measure the evolution of the Dartmouth College network from 2005 to 2009. The specific properties that we measure include changes to (1) our inventory of object groups and access-control lists, (2) our similarity graphs of object groups and ACLs, and (3) our report on the usage of ACLs in `interface` commands.

Set-up

In order to run this experiment, we used the CVS repository provided by Dartmouth Computing Services to create five corpora of network device configurations. The first (second, third, fourth, fifth) corpus contained the core network device configurations as of June 1, 2005 (2006, 2007, 2008, 2009). We then ran each of our previous three experiments on the corpora. We view these five corpora as a multiversioned corpus.

Results and Interpretation

We now present the results of how our inventory of object groups and access-control lists, our similarity graphs of object groups and ACLs, and our report on ACL usage evolved.

Evolution of Object Group and ACL Inventories: Table 7.3 shows the evolution of the number of object groups and ACLs within the Dartmouth core network from 2005 to 2009.

Table 7.3 shows us that object groups only started to be defined within the Dartmouth Core in 2008. In addition, the average number of ACLs has steadily hovered at around 7, but the number of ACLs themselves has more than tripled in 5 years.

Object Groups in the Dartmouth Core: 2005-2009		
year	**number**	**size (min/avg/max)**
2005	0	0/0/0
2006	0	0/0/0
2007	0	0/0/0
2008	6	2/4.0/6
2009	117	2/4.0/21

ACLs in the Dartmouth Core: 2005-2009		
year	**number**	**size (min/avg/max)**
2005	18	2/6.0/39
2006	34	2/8.0/80
2007	39	2/7.0/39
2008	52	2/6.0/39
2009	64	2/7.0/39

Table 7.3: From 2005 to 2009, the number of object groups increased from 0 to 117 with the largest being 21 lines. In addition the number of ACLs increased from 18 to 64.

Evolution of Similarity Graphs of Object Groups and ACLs Table 7.4 shows the evolution of the similarity graphs for object groups and ACLs within the Dartmouth core network from 2005 to 2009. As mentioned before, an edge between two object groups or ACLs indicates that the parse trees of those constructs are at least 60% similar via tree editing distance metric. In this manner, we literally measure security policy evolution.

Table 7.4 shows us that although the number of ACLs increased more than 3 times in 5 years, the number of ACLs that were structurally similar hovered between 2 and 10. This indicates that most ACLs are less than 60% structurally similar to one another. Given this, network administrators must have tools to keep track of lots of small ACLs that are on average 7 lines long according to our 2009 inventory of ACLs.

Object Groups in the Dartmouth Core: 2005-2009					
year	**number**	**clusters**	**3-clusters**	**2-clusters**	**unclustered**
2005	0	0	0	0	0
2006	0	0	0	0	0
2007	0	0	0	0	0
2008	6	0	0	0	0
2009	117	100	4	9	87

ACLs in the Dartmouth Core: 2005-2009					
year	**number**	**clusters**	**3-clusters**	**2-clusters**	**unclustered**
2005	18	17	0	1	16
2006	34	31	0	3	28
2007	39	36	0	3	33
2008	52	49	0	3	46
2009	64	58	2	2	54

Table 7.4: From 2005 to 2009, we see that the number of ACLs increased from 18 to 64 but that the number of ACLs that are structurally similar has remained relatively stable since 2006 when there were only 34 ACLs defined in the core network.

Evolution of ACL Usage: Table 7.5 shows the evolution of the number of unique ACLs in the core network that were used on interfaces in the core network.

In Table 7.5 we see that the number of unique ACLs within an `interface` command remained relatively stable even though the number of ACLs more than tripled. Furthermore, there was only one ACL within a `class-map` command. In other words, even though there are many ACLs defined in the core network, a relatively small proportion are utilized in interfaces. More precisely, a relatively small proportion of ACLs are applied to interfaces using the `ip access-group` syntax.

7.2.5 Section Summary

This section demonstrated how XUTools enables new computational experiments on structured texts whose results give practitioners new capabilities. Furthermore, these new capabilities directly address the core limitations of change summarization and measurement in network configuration management that we observed in our fieldwork.

ACLs Used in the Dartmouth Core: 2005-2009			
year	number	interfaces	class-map
2005	18	6	0
2006	34	14	1
2007	39	19	1
2008	52	20	1
2009	64	21	1

Table 7.5: From 2005 to 2009, the number of ACLs defined in the *core* increased more than threefold while the number of unique ACLs applied within an `interface` command remained relatively stable.

7.3 Evaluation

In this section, we evaluate the novelty and utility of these capabilities by reporting anecdotal feedback that we received from real-world network administrators and auditors after we demonstrated these capabilities.

The goal of this chapter is to demonstrate that our XUTools gives system administrators and auditors practical capabilities by enabling new computational experiments on structured texts. We enabled these new experiments by directly addressing the three core limitations of security policy analysis that we introduced in Chapter 2.[3]

[3]Furthermore, these limitations are closely aligned with research barriers to national strategic research goals for smart grid control networks.

- There is a gap between the tools available to system administrators (RANCID, Cisco IOS `include`) and the languages they use to represent security policy artifacts (*Tool Gap Problem*).
- Network administrators and auditors need to be able to process policies at multiple levels of abstraction and they currently cannot (*Granularity of Reference Problem*).
- Practitioners need a feedback loop for network security policies and artifacts so that they can understand how their network evolves (*Policy Discovery Needs Problem*).

7.3.1 General Feedback from Practitioners

The majority of system administrators and auditors with whom we spoke were from the domain of the power grid. Despite this, however, many of these individuals have a general knowledge of network system administration and audit. We demonstrated these capabilities during the 2012 TCIPG Industry Day in the form of a video demonstration that played during breaks, a poster session in the evening, and a live presentation to the TCIPG EAB. The video of our demonstration of these capabilities may be found at http://youtu.be/onaM_MS6PRg.

Capabilities Interface: Shortly after we agreed to give a demonstration of our XUTools based capabilities, researchers pointed out that a Command-Line Interface (CLI) would not provide an interesing demonstration, nor would a CLI be usable by many auditors. In order to address this limitation, we developed a visualization layer for our first capability, inventory network security primitives. We will discuss this visualization layer in more depth in Chapter 8.

Feedback from Edmond Rogers

We originally developed these capabilities in response to the recommendations of Edmond Rogers, a utility expert who maintained power-system control networks for a living. For years, Edmond had to keep a major utility's corporate and power-control computer networks secure. If there was an incident, he would have lost his job and people would lose electric power. Therefore, we take his advice very seriously and this is why he is the "utility expert" for TCIPG. In addition, Edmond participates in several audits of power-control system networks per year.

First, Edmond thought the capability to be able to inventory and measure the similarity between security primitives on Cisco IOS devices was useful and could save auditors a lot of time. In particular, he said that many times "similar devices have object group structures that are meant to be identical across the enterprise" and that the tools could show subtle changes in these structures. In Section 7.2, we already demonstrated the ability to find subtle changes in near-identical structures using a similarity graph.

In addition, Edmond thought that our first capability, the ability to inventory security primitives, could be useful to actually measure the interplay between a network's attack surface and VPN and ACL settings. In the experiment sketch he provided, this process would involve measuring the attack surface, grouping network devices according to the size of ACLs that they contain (via our first and second capabilities), and then highighting atomic differences that may differentiate one ACL from similar ACLs (via `xudiff`). In this manner, one could use XUTools to systematically explore the relationship between attack surface and ACL configuration.

Feedback from Bryan Fite:

Bryan Fite is a Security Portfolio Manager at British Telecommunications (BT). In addition, he runs the DAY-CON security summit [37] and PacketWars, an informa-

tion warfare simulation [93]. When I spoke with Bryan about our XUTools-based capabilities, he thought that the abilities to inventory the number and size of security primitives and to measure properties of these primitives over time were useful capabilities. These capabilities are useful because the auditors and administrators that Bryan encounters when he's on an audit currently don't have an easy way to know what their "normal" security posture looks like. In other words, Bryan echoed the need for a feedback loop for security posture. When we demonstrated our tools to Bryan during a break, he wondered whether there was an interface between our tools and change management systems like ChangeGear [22] or Remedy [106]. Currently, there is no such interface; we consider his suggestion potential future work.

Feedback from Himanshu Khurana:

Himanshu Khurana is a senior technical manager for the Integrated Security Technologies section at Honeywell Automation and Control Systems Research Labs. He serves on the TCIPG EAB. He saw our demonstration of the ability to inventory network security primitives and our description of how we can measure security primitive evolution. After the demonstration, he asked how our system compares to Splunk, a highly-scalable indexing engine [121]. Although Splunk allows practitioners to search a variety of information by extracting or learning key/value pairs, it does so via regular expressions. In contrast, XUTools can be used to index information for search in terms of structures in a context-free language.

7.3.2 Related Work

We now evaluate our capabilities in light of the current approaches to summarize and measure change within network configuration files. We discussed these approaches earlier in Chapter 2. For the sake of clarity, we evaluate our capabilities relative to both the current *state of the practice*–techniques and procedures currently employed

in production network environments–and the current *state of the art*–methods that researchers have developed to improve change summarization and measurement.

State of the Practice: Network administrators use the Really Awesome New Cisco configuration Differ (RANCID) to track how network device configurations change. RANCID is helpful, but because it records changes in terms of lines, practitioners cannot see the big picture as to how the network is changing.

In order to see the big picture, practitioners may use change-management products such as ChangeGear [22] and Remedy [106] that are ticketing systems. However, they rely upon manual documentation to record changes.

Finally, information indexing systems such as Splunk [121] do allow practitioners to get a big-picture view of how data changes over time. Practitioners can teach Splunk how to extract fields from their data using the Interactive Field Extractor, by specifying templates, or by parsing XML element/element value pairs, but in all of these cases the structures reduce to key/value pairs matched via regular expressions. Splunk's indexing capability is limited however because many high-level languages have context-free structures and some have language constructs that are even more general.

Our XUTools-enabled capabilities improve upon the state of the practice and measure the evolution of network security primitives over time. Practitioners can see the big picture and avoid manual documentation as the trusted source of change documentation.[4]

State of the Art: Our parse-tree based model of Cisco IOS provides a more general, theoretical framework for the stanza-based (blocks at the top indentation level of Cisco IOS) analyses by Plonka and Sung [98, 126]. Our XUTools approach provides a library of operations on context-free structures in general that is backed by

[4] In Chapter 2 we saw that manual changelogs often are insufficient and unreliable.

language-theoretic underpinnings. Furthermore, our approach uses tools that are freely available and these tools and approaches should generalize to other types of network configurations and logs that contain high-level, context-free structures (such as Juniper router configurations).

Plonka et al. studied the evolution of router configurations and concluded that future services built for configuring network devices would be "well-advised to cater specifically" to the management of `interface` stanzas [98]. Our capabilities heed this advice in that they help practitioners to manage the ACLs associated with interfaces.

Kim et al. recently did a very complete longitudinal study of the evolution of router configurations, firewalls, and switches on two campuses over five years. They looked at the frequency of configuration updates, and identified correlated changes among other things [71]. Our approach is complementary, for our parse-tree based approach allows one to view network evolution at a variety of levels of abstraction ranging from the file and device-level views of Kim et al. down to the changes to a specific ACL, object group, or interface.

In their NSDI 2009 paper, Benson et al. present complexity metrics to describe network complexity in a manner that abstracts away details of the underlying configuration language [8]. Our investigations are based upon tools designed for practitioners and auditors alike to run on their own networks. These investigations do not abstract away the details of the configuration language, but performs a change analysis *in terms of the structures defined by the configuration language.* We argue that this makes for a more intuitive interpretation of results.

7.3.3 Capability-Specific Evaluation

We now discuss some specific evaluation points regarding our first three capabilities.

Inventory of Network Security Primitives

The first capability directly addresses the gap between the high-level languages in which network configuration files are written and the low-level languages (lines, key-value pairs, files) on which tools available to practitioners operate. In addition, this capability allows practitioners to operate on language constructs in terms of multiple levels of abstraction. An inventory of network security primitives allows practitioners to quickly drill down through the hierarchical organization of the network and configuration language and identify *important* network security primitives.

We need to accommodate other, more sophisticated measures of *importance* than the line-count based approach suggested by our utility expert. For example, at a recent demonstration of our tools to practitioners in the power grid, one person wondered whether it would be possible to count the number of IP addresses that are associated with an ACL. Since IP addresses may be represented using a subnet syntax (for example 0/24) or listed one at a time, we would want a special counting function to perform this kind of analysis.

Similarity of Network Security Primitives

The second capability directly addresses all three core limitations of security policy analysis.

Our first approach was to find similarly-named ACLs within our ACL inventory and then compare those ACLs using our `xudiff`. This allows administrators to discover slight variations in similarly-configured ACLs. We observe that the manual nature of our approach could be improved by using string metrics to find similarly-named security primitives automatically.

Our second approach successfully detected roles that had very different names but were quite similar. More generally, however, the second capability demonstrates the practical application of clustering as an operation on strings in a language. We now

evaluate the similarity measure, as well as the construction of the similarity graph.

Given a set of roles as our data points, we constructed a similarity matrix using a similarity measure based on the Zhang and Shasha tree edit distance. We now describe our similarity measure between corpus elements. For a pair of corpus elements x_i and x_j, we parse the value of their `text` fields into parse trees T_i and T_j. We then compute the tree edit distance between these two trees using the unit cost metric for node edit operations. Finally, we normalize the tree edit distance by the maximum number of nodes in either of the two parse trees $(\max(|T_i|, |T_j|))$. This similarity measure is not a distance metric. Nonetheless, our measure does allow us to cluster language constructs in a manner that favors elements with a similar number of nodes in their parse trees.

In contrast, we could develop a similarity function that is a metric but the metric we considered would not cluster elements whose parse trees have a small number of nodes. If we normalized the tree edit distance between T_i and T_j by the number of nodes in the largest parse tree of the corpus, then we would have a similarity metric. Unfortunately this metric does not produce the clusters that we want.

For example, assume that corpus elements x_i and x_j are object groups with different names, but matching content. As a result of their close similarity, their corresponding parse trees, T_i and T_j, have an tree edit distance of (1). Also assume that each of these parse trees consist of 4 nodes. These are realistic assumptions. In Table 7.3, we see that on average, there were 4 lines per Object Groups in the Dartmouth Core network as of June 1, 2009. By the same table, the largest parse tree in the corpus of roles has 21 lines.

If we normalized the tree edit distance (1) for T_i and T_j by the size of the largest parse tree (21) in the whole corpus, we would get a similarity score of approximately 0.05. Even though x_i and x_j are the same except for their names, they would not be included in the similarity graph because 0.05 is much less than our similarity

threshold (0.6) for edges. We chose this threshold empirically. We generated clusters using thresholds that divided the interval from 0 to 1 into tenths (0.1, 0.2,...0.9). We then chose the threshold based on which of the clusters captured useful similarity measures without making the graph too sparse or dense.

We should note that the similarity-graph approach to clustering elements in a corpus sets the foundation for more sophisticated clustering algorithms such as spectral clustering. The similarity-threshold approach we described above, if rephrased in terms of a dissimilarity measure, is the ϵ neighborhood approach to constructing a similarity graph [136]. Furthermore, the similarity graph we describe in the example above has multiscale structure and there exist methods that take into account the "heterogeneity of local edge weight distributions" [45]. We leave this as potential future work.

Usage of Network Security Primitives

Our third capability allows practitioners to measure which of the defined security primitives are actually used. Specifically, we measured how many unique ACLs were actually applied to network interfaces and class-maps. We also noticed that, over time, the number of unique ACLs applied within an interface remained relatively stable even though the number of ACLs more than tripled. During this experiment, however, we noticed two limitations of our approach.

First, we can use `xuwc` to count the total number of ACLs applied to a network interface, but we cannot use `xuwc` to count the number of *unique* ACLs applied to a network interface. In order to get the unique ACLs, we had to use `xugrep` in concert with traditional Unix tools such as `uniq`. We can pipeline our XUTools with traditional Unix tools because `xugrep` can output corpus elements to lines via the `-1` option. We consider this limitation of `xuwc` to be acceptable however, because `wc` also cannot count the number of unique bytes, words, lines, or characters in a set of

input files.

Second, we noted in our results that we measured the number of ACLs applied to interfaces using the `ip access-group` syntax. There is another command used in Cisco IOS to apply an ACL to an interface but this command occurs outside the scope of the interface block. In the `ip access-group` syntax, the hierarchical structure of the command syntax encoded the application of an ACL to its containing interface. In this other syntax, the relationship between applied ACL and interface is no longer encoded in the parse tree. Certainly, we could craft a XUTools based script to parse out these commands, but this highlights that our tools are at their best when they operate on relations encoded by the parse tree edges.

7.4 Conclusions

Network administrators and auditors can use XUTools to understand changes to Cisco IOS constructs such as object groups, ACLs, and interfaces at a variety of levels of abstraction that range from an entire network down to a single IP addresss. Although our approach has a language-theoretic foundation, the practical implications of our tools means that practitioners can inventory, measure similarity, and see the usage of high-level language constructs in network configuration files. Finally, since these language constructs have names that persist across multiple versions of a configuration file, we can use those constucts as units of analysis to quantify network evolution.

Chapter 8

Future Work

This chapter introduces several potential future research directions that build directly upon our XUTools. Section 8.1 describes our ongoing efforts to apply XUTools to the domains of X.509 PKI, terms of service policies, and the electrical power grid. In Section 8.2 we introduce several new application domains that would broaden the applicability of XUTools. Finally, we propose several extensions to XUTools and our theoretical toolbox for structured text analysis.

8.1 Ongoing Research

In this section we describe ongoing research that applies XUTools to the domains of X.509 PKI, terms of service policies, and the electrical power grid.

8.1.1 Application of XUTools to X.509 PKI

PKI policy analysis and management, by its very nature, tries to understand changes to security policies across time at a variety of levels of abstraction. XUTools enables practitioners to *measure* security policy evolution *directly* from security policies and security policy artifacts.

As discussed in Chapter 2, security policy comparison is a fundamental operation in several X.509 PKI processes that include PKI Compliance Audit, IGTF Accreditation, and Policy Mapping to Bridge PKIs. Although policy comparison is a fundamental operation in all of these important X.509 processes, it remains a manual, subjective process and these drawbacks make it inconsistent. Compliance audits, accreditation procedures, and policy mapping decisions are difficult to reproduce because they are so dependent upon auditors' individual observations.

Problems with PKI Policy Comparison

In Chapter 2, we described three core limitations of security policy analysis that negatively impact the PKI policy comparison process. First, there is a gap between the traditional text-processing tools (Adobe Acrobat Reader, Microsoft Word's Track Changes) and the languages used in security policies (RFC 2527 and 3647) and this gap forces policy operations to be largely manual. Second, practitioners lack tools to process policies on multiple levels of abstraction ranging from large global federations, to individual organizations, to Certificate Policies (CPs) and provisions within those CPs. Finally, practitioners lack a feedback loop by which they can measure security policies and how they evolve. For example, when the IGTF changes an Authentication Profile (AP), member organizations have 6 months to re-certify that they are compliant with the new profile and volunteers must perform this manual verification.

Results from one of our previously-published pilot studies [139] suggest that the changelog-based IGTF accreditation process is insufficient to measure how policies change through time. For example, our previous study of 13 International Grid Trust Federation (IGTF) member organizations revealed 5 organizations with at least one *reported* change in their changelogs for which there was no *actual* change in the policy. Out of a total of 178 reported changes, 9 of those changes corresponded to no actual change. Of the 94 of 178 changes that claimed a major change, we found 5 that

were logged but never performed [139]. Current practice does not suffice for effective management.

Our Ongoing Studies

Our pilot study for this ongoing research focused on IGTF Accreditation. Recall from Chapter 2 that the IGTF sets standards for PKI certificates used to authenticate to different grids worldwide and these standards are documented in an Authentication Profile (AP). The IGTF consists of three member organizations, the European Union Grid Policy Management Authority (EUGridPMA), the Asia-Pacific Grid Policy Management Authority (APGridPMA), and The Americas Grid Policy Management Authority (TAGPMA). These policy management authorities set their own standards for member Certificate Authorities (CAs) in a manner that is consistent with the IGTFs standards.

In our ongoing research, we are exploring ways to measure security policy evolution using our XUTools. As a proof-of-concept, we selected a set of 4 EUGridPMA CAs from our PKI Policy Repository, a multiversioned corpus of roughly 200 CP/CPSs that we assembled.[1] More information about our PKI Policy Repository may be found in Appendix A. Table 8.1 summarizes the number of versions recorded of each CA's policy and the date ranges they span.

	EUGridPMA CP/CPSs			
	AIST	CALG	UKeSci	ArMes
versions recorded	6	7	8	7
date range recorded	9/04 - 9/09	3/07 - 1/09	7/02 - 11/07	3/04 - 6/09

Table 8.1: We selected a set of 4 CAs from the EUGridPMA from our PKI Policy Repository. For each of the policies in our corpus, the table shows the number of versions recorded and the date range spanned by these versions.

[1]See http://pkipolicy.appspot.com/.

Our hypothesis (as with network configuration) is that CAs and analysts will benefit from high-level languages for textual analysis. In this research we want to use XUTools to measure the evolution of IGTF CP/CPS policies. As a result of this work, we have developed a new cost metric for tree node edit operations that incorporates parse-tree specific information (the production associated with a node) to produce a matching that is better suited to our notion of similarity than the unit cost metric provided by Zhang and Shasha [151].

8.1.2 Application of XUTools to Terms of Service Policies

An important part of evaluating a web-based service is to understand its terms and conditions. Terms and conditions are described in a service's terms of service agreement. Casual readers, however, often agre to the terms without reading them. Despite this, enterprises as well as individual consumers need to be able to compare terms of service for web-based services [96].

Enterprise-level cloud customers need to be able to compare different cloud provider offers in order to decide which services are consistent with their requirements. For example, in a 2010 *IEEE Security and Privacy* article, Pauley describes a methodology to compare the security policies of various cloud providers [96]. Furthermore, the European Network and Information Security Agency (ENISA) did a cloud computing risk assessment and stated that customers need to be able to compare different provider offers in order to decide which services are consistent with their requirements [20].

Individual consumers of cloud services should also be able to compare terms of service quickly and reliably to comprehend the risks of using a service. The Electronic Frontier Foundation (EFF) recognizes the importance of understanding changes to security policies and so built a policy comparison tool, the TOSBack [130]. Google also provides a way to compare archived versions of their security policies.

Problems with Terms-of-Service Policy Comprehension

The current, mainstream mechanisms that companies use to inform users of their terms of service are problematic with serious consequences. Very recently, an article in *TechCrunch* appeared that quipped "I agree to the terms of service" has long been called the "biggest lie on the internet" [42]. A new project called TOS;DR reports manual summaries of terms-of-service agreements so that users have more choice. Terms-of-service agreements can have real legal consequences; in October 2012, a court ruled that Zappos' terms of service was completely invalid for not forcing its customers to click through and agree to them. In addition, Zappos' terms say that it can change the agreement at any time and the courts have invalidated contracts on this basis before [128].

Changes to policy matter. In March 2012, Google changed its privacy policy so that private data collected via one Google service can be shared with its other services such as YouTube, Gmail, and Blogger [53]. The EU justice commissioner, however, believed that those changes violated European law.

Our Ongoing Studies

We propose to use XUTools to help enterprise and consumer-level customers compare terms of service policies and understand how they change. For example, customers could agree to some version of a terms of service contract and after that time, they could agree (or even disagree) with changes to that policy.[2] A customer interested in how Facebook shares third party information could also just look at changes to that particular section of policy.

In ongoing research, we want to explore how to measure terms-of-service policy evolution directly using XUTools. As a proof-of-concept, we downloaded 24 versions of the Facebook policy recorded by EFF's TOSBack between May 5, 2006 and Decmber

[2]Thanks to Dan Rockmore.

23, 2010. For breadth, we have conducted pilot studies on three kinds of terms-of-service policies: privacy policies, music terms-of-service policies, and cable terms-of-service policies. More information about this corpus may be found in Table 8.2.

	Privacy Policies			
	Apple	EFF	Facebook	Gmail
versions recorded	7	3	24	4
date range recorded	6/07 - 6/10	2/09 - 6/11	5/06 - 12/10	3/09 - 2/10

Table 8.2: We selected a set of four multi-versioned privacy policies recorded by the EFF Terms of Service tracker. For each of the policies in our corpus, the table shows the number of versions recorded and the date range spanned by these versions.

Our hypothesis is that service consumers will benefit from using high-level languages for textual analysis. Already, we have used our XUTools to identify November 19, 2009 as a major event in the evolution of Facebook privacy policy. In fact, the Wikipedia article "Criticism of Facebook" has an entire subsection devoted to this event. The article states that in November 2009, Facebook's newly proposed privacy policy were protested and even caused the Office of the Privacy Commissioner of Canada to launch an investigation into Facebook's privacy policies [36]. This initial study suggests that XUTools change measures seem to be able to pinpoint important events in the *big picture* history of a terms-of-service policy. Moreover, we can use the `xudiff` to *drill down* and find specific policy changes.

8.1.3 Application of XUTools to the Power Grid Data Avalanche

As discussed in Chapter 2, the smart grid will increase the stability and reliability of the grid overall with vast numbers of cyber components and many of these components will generate data. The smart grid of today is already large, serving roughly 160 million residences in the United States and non-residential customers as well. At the

2012 TCIPG Industry day, one gentleman said that he worked for an Investor Owned Utility (IOU) that have 700 substations and 7 million residential customers. Today and in the future, the smart grid has and will have a large number of devices that will need to be configured and produce and log data.

Failure to manage this data has real consequences. As noted by NISTIR 7628, "increasing the complexity of the grid could introduce vulnerabilities and increase exposure to potential attackers and unintentional errors" [58].

High-Level Research Challenges

As discussed in Chapter 2, high-level research challenges for smart grid cybersecurity stated by the DOE and NIST directly align with our three core limitations of security policy analysis.

First, the Roadmap says that practitioners need the capability of being able to operate on "vast quantites of disparate data from a variety of sources". We designed our XUTools to operate on a vareity of data formats in terms of their hierarchical object models and many of the new smart grid data formats have a hierarchical object model.

Second, the Roadmap also says that practitioners need the capability of being able to operate on data at a variety of levels of granularity. Our XUTools represent hierarchical structure as parse trees and in Chapter 3, we discussed how parse trees allow us to process text at multiple levels of abstraction.

Third, the program director of the Smart Grid and Cyber-Physical Systems Program office stated in his Smart Grid Program Overview that we need new measurement methods and models for the smart grid [4]. We designed XUTools to be able to measure properties of texts and how those properties change over time.

Our Ongoing Studies

We now present a few potential applications of XUTools in the domain of the electrical power grid. This work was already presented to the External Advisory Board of our Trustworthy Cyber Infrastructure for the Power Grid project and we will be presenting this work to funders in Washington D.C. in the near future.[3] We first discuss capabilities from Chapter 7 and how they specifically address needs of the electrical power domain. We then conclude with several additional research directions.

Inventory of Network Security Primitives: XUTools processes data at multiple levels of granularity to pinpoint complex network security primitives. The demonstration dataset for this capability was an artificial but realistic collection of network configurations from a major Investor-Owned Utility (IOU). The data was constructed based upon the topology observed in this utility's network. The IP addresses were anonymized and some hosts were removed. Specifically, we want to be able to inventory the security primitives and pinpoint *important* primitives. This capability, developed in Chapter 7, addresses several vulnerabilities and regulations in the power domain. If done efficiently, we save auditors time and reduce audit cost (NISTIR 6.2.3.1).[4] When performed to catalog roles—groupings of users, devices, and protocols—defined in access-control policies, we make roles easier to manage (NISTIR 7.3.23) [58]. Finally, this capability helps to address the need for better baseline configuration development and comparison (CIP 010-1) [88].[5]

[3]For a video of our presentation, please go to `http://youtu.be/onaM_MS6PRg`.

[4]*The National Institute of Standards and Technology Interagency Report (NISTIR)* discusses guidelines for smart grid cybersecurity. This three-volume report includes a high-level architectural overview of the smart grid and enumerates threats to its security [58].

[5]The North American Reliability Corporation (NERC) developed a set of security standards for Critical Infrastructure Protection (CIP). These guidelines, while not regulations, capture the best practice in basic security controls for the smart grid today according to Jeff Dagle at *Pacific Northwest National Laboratory (PNNL).*

Evolution of Security Primitives: Our XUTools measure how security primitives change. We want to be able to generate changelogs directly from router configurations and measure how network configurations have changed over time. As discussed in Chapter 2, writing changelogs is a time-consuming, error-prone process. The relevance of logs may change over time and auditors may want information at a different level of abstraction than utilities. Utilities and auditors also may want to measure how security primitives have changed to measure both how and how frequently roles change in an access-control policy. These capabilities address several vulnerabilities and regulations that include "Inadequate Change and Configuration Management" (NISTIR 6.2.2.5), audit log forging, and several NERC CIP requirements [58].

We designed XUTools to operate on hierarchical object models in general, however, and there are several other possible applications of these tools to the smart grid. For example, we could use XUTools to compute a common communications interface in substations by defining equivalence classes between corpus elements in the Common Information Model (CIM) and IEC 61850 languages or by comparing IED Configuration Descriptions (ICDs) as defined by IEC 61850.

8.2 Additional Problem Scenarios

Our XUTools have applications far beyond the domains we discussed in Chapter 2. In this section, we will motivate new research in three new problem domains that could benefit from XUTools.

8.2.1 Healthcare Information Technology

Information Technology (IT) plays an increasingly important role within healthcare delivery. For example, hospitals are increasingly deploying Electronic Medical Records (EMRs) in order to improve patient care and administrative tasks by

making health records easier to maintain and supplement clinician's decision-making processes via Clinical Decision Support (CDS).

In practice, however, these high-level goals for EMR systems are accompanied by usability issues after a system is deployed. For example, patient records may be corrupted by copy-and-pasted content. In addition, CDS systems bombard clinicans with too many warnings and options to be useful.[6] The task faced by clinicians is to be able to analyze and manage these corpora of messy medical data. The consequences of failing to properly manage patient data includes misdiagnosis and even patient death.

Limitations of Textual Analysis

In fact, many of the current problems with EMR are symptoms of our three core limitations of textual analysis. First, there is a gap between existing EMR systems and the workflows used by clinicians. Second, clinicians need to be able to reference and analyze EMRs on multiple levels of abstraction. Finally, clinicians need to be able to measure properties of medical records and how those properties change over time.

Gap Between EMRs and Clinical Workflows: Clinicians want to be able to determine how they organize and communicate data to one another and they can't due to EMR homogeneity. This homogeneity leads to problems due to unmapped reality. Clinicians communicate with one another using one mental model, but when they map that model into a commercial IT product, problems due to system language being too coarse or fine-grained relative to their mental models occur [119].

[6]Email with Sean W. Smith and Ross Koppel. Sean W. Smith is my advisor and I owe him vast quantities of beer at the very least. Ross Koppel is an Adjunct Professor of Sociology at the University of Pennsylvania. His research interests primarily focus on healthcare information technology and its use in clinical environments. He has recently co-authored a book on this topic: *First, Do Less Harm: Confronting the Inconvenient Problems of Patient Safety* [74].

Hospital data and workflows are inherently heterogeneous and yet commercial medical products push homogeneity. Hospital data and workflows are inherently heterogeneous from the level of the state in which the hospital is located, to the hospital organization, to the departments in which clinicians are located, to the individual patients which they see.

At the level of the state, legislation varies. For example, DHMC must accommodate patients from both New Hampshire and Vermont who have varied right to their information. In New Hampshire, medical information contained in medical records is deemed the property of the patient under Title X of the NH Patients' Bill of Rights [124].

At the level of the hospital, workflows and procedures vary. The head of medical informatics at one local hospital stated that in medicine, if you understand how one hospital works, then you understand how one hospital works.

At the level of the departments in which clinicians operate, birthing, and cancer treatment areas all have different procedures and characteristics unique to the services that they perform to care for the patient.

At the level of the individual doctor and patient, there are unique circumstances that affect how data is recorded. For example, a patient may have a mild form of a disease but that disease is recorded as being much worse so that insurance covers the condition. In contrast, a patient may have a horrible disease, but the doctor records the disease as something more mild in order to prevent embarrassment.

In contrast to this heterogeneity, however, the DHMC uses an EHR product that accounts for 40% of the EHR records in the United States. Heterogeneity, although a characteristic of the medical environment, is a second thought in these products; customization may be purchased after the base installation. This is backwards and leads to problems. Geer's IT monoculture argument outlines implications of heterogeneity for security [51].

Medical Records have Multiple Levels of Abstraction: Clinicians write and interpret medical records at multiple levels of abstraction. Clinicians often must fill out fields in an EMR system with a drop-down menu of valid field values. Unfortunately, the level of abstraction at which these menu options are expressed, may not be appropriate for every clinician. A story related by Ross Koppel underscores this point. A doctor suspects that one of his patients has stomach cancer and so wants to refer the patient to a specialist. In order to refer the patient to a specialist, the doctor must scroll through a list of "scores" of different stomach cancers and pick one. The *stomach cancer* options are too specific and so the doctor must select one of the options (but probably the incorrect specific diagnosis) [74].

Clinicians Need to Measure Evolving Medical Records: Finally, clinicians want to be able to measure and operate upon medical records and how they change through time but currently cannot.

First, clinicians want to be able to measure properties of medical records. For example, patient records may be corrupted by massive copy-and-paste and clinicians want to be able to pinpoint where this occurred.

Second, clinicians want to be able to measure and analyze how medical records evolve. The head of informatics at a local hospital thought that the *timeline* approach to medical data that Google Wave offered was a very compelling view of an EHR [55]. Timelines are appealing to practitioners because medical workflows and patient care inherently unfold over time. Furthermore, if EMRs are going to deliver on the ability to statistically evaluate medical procedures in terms of patient outcomes, clinicians will need the ability to measure the evolution of patient care from EMRs.

Summary

Although hospital data and workflows are inherently heterogeneous, commercially-available EMRs push homogeneity and this leads to problems that threaten patient safety and make clinicians' jobs harder. Clinicians solve problems everyday by creating their own language to specify the relevant details and abstract away the irrelevant details of a problem. Unfortunately, modern EMRs do not allow clinicians to express these unique approaches in a way that we can systematically process. XUTools can implement several new capabilities that help practitioners create and process medical information at multiple levels of granularity, and pinpoint copy-and-pasted information as a source of potential corruption or reinterpretation.

8.2.2 Legislation and Litigation

Lawyers and legislators must understand how legal documents change. Since many people simultaneously edit these documents, it becomes practically infeasible to manually detect changes between versions. Additionally, both the size of the law and the restrictions on the time provided to read the legal document make understanding how legislation changes practically impossible.

We have seen an example of this quite recently in which the Supreme Court was unable to even read the *Patient Protection and Affordable Care Act* to determine whether it was constitutional or not [123]. Furthermore, when we looked at the first and second-level table of contents of this bill, the tables of contents were inconsistent. In other words, the structure of the bill was not even valid. This example indicates that we need new methods to help legislators write and evaluate law.

Already we have received feedback on applying our XUTools to this domain from several practitioners. Jon Orwant, who is in charge of Google Books, is particularly interested in our research and visualizations of legislation. Furthermore Dan Rockmore and his colleague Michael Livermore, a lawyer at NYU's Institute for Policy

Integrity are both interested in ways to measure the evolution of legal documents. The volatile nature of these documents, combined with their size, pose challenges that make legal documents a logical next step that builds upon our work on X.509 PKI policy analysis.

8.2.3 Operating Systems and Trusted Hardware

Several different aspects of Operating Systems might benefit from a XUTools-based approach. We discuss how problems in memory management and trends in research are symptoms of the three core limitations of textual analysis.

First, there is a gap between the low-level units used by traditional memory management schemes and the high-level data structures that applications use. We see this in *Language-Theoretic Security (LangSec)*, a research group based out of our lab at Dartmouth that hypothesizes that system vulnerabilities are consequences of ad hoc-programming of input handlers [12]. One direction of LangSec research is that there is a gap between the paging system used by the operating system and the language of the Application Binary Interface (ABI). The semantic gap has consequences that include the inability to enforce meaningful memory protections.[7]

Second, in memory management there is a need to be able to reference and operate on regions of memory at a variety of levels of granularity. Research into new paging schemes such as Mondrian Memory attest to this need [143].

Third, systems need to be able to measure different constructs within memory and how they evolve. Consider trusted boot's Trusted Platform Module (TPM). During trusted boot, hashes of software binaries (memory images) are used to evaluate whether or not an image is trustworthy [118]. As we noted before in our discussion of TripWire, hashing images ensures integrity but it does not indicate *how* software has changed. A XUTools-based approach might be able to not only determine *whether*

[7] This is ongoing work in the TrustLab and is partially funded by a grant from Intel.

software has changed, but *how* it has changed. For example, by aligning a Merkle tree with a parse tree structure one could determine which parts of a configuration (and maybe even a binary) changed without having to know its contents. A literature review would need to be done in this area to determine novelty.

Furthermore, the ability to measure different constructs within memory and how those constructs evolve has applications to memory forensics. Our work was already cited in the context of this topic in a paper by Garfinkel et al. [49].

8.3 Additional XUTools Extensions

We now motivate several extensions to our XUTools. Several of these extensions require new theoretical tools. We organize the discussion by XUTools modules.

8.3.1 Version Control

If we can extend our XUTools to instantiate corpora from version-control systems, then we could measure and analyze how high-level languages evolves in the real-world.

For example, if we used `xudiff` as a difference engine for version-controlled configuration files, then we could provide useful debugging information to a system administrator when used in concert with bug reports.[8] In essence, `xudiff` would give us an easy way to identify the most volatile language constructs within a set of configuration files and this is a useful tool for debugging. A workshop at *LISA 2011*, *Teaching System Administration*, stated that the parts of the system that break down historically are the first places that administrators should look for bugs. Our tools would allow practitioners to pinpoint the regions of configuration that were changed when the bugs were first reported. We also think such an application of our tools would help organizations understand legacy networks acquired via mergers.

[8] Feedback from Dartmouth Lake-Sunapee Linux Users Group (DLSLUG).

8.3.2 Grammar Library

The utility of our XUTools depends on the availability of grammars for languages in which practitioners are interested. One way to increase the size of our grammar library is to write a conversion tool from languages used to specify grammars.[9] This strategy would allow us to reuse work done to write grammars for traditional parsers such as Yacc and ANTLR, cutting-edge configuration tools like Augeas, and grammars for XML vocabularies such as XSD and RELAX-NG [80, 95, 105, 146, 148].

Finally, practitioners might be able to parse a text written in one language using a grammar for a very *similar* language. For example, the emacs editing mode for bash highlights Perl syntax reasonably well. This *approximate matching* strategy could be a mechanism for XUTools to handle languages for which there is no grammar in the grammar library.[10]

8.3.3 Distributed Parsing

One future direction could be to consider distributed parsing algorithms. Professor Bill McKeeman had done research in this area previously and it would allow our XUTools to handle larger data sets.

8.3.4 Distance Metrics for XUDiff

In the future, we want to broaden XUTools to include distance metrics beyond simple string and tree edit distances. In fact, there are several theoretical tools from machine learning and information retrieval that we can apply to measure similarity between corpus elements.

Traditional approaches in information retrieval use distributional similarity to measure distance between documents. The documents often are defined in terms of

[9] Thank you Tom Cormen

[10] Thanks to David Lang and Nicolai Plum at the LISA 2011 poster session.

physical schemes such as number of words, lines, or pages. The documents are then processed into a vector of frequencies in which every element in the vector corresponds to a document token (a word or lemma). These document vectors are then clustered based upon simple vector distance metrics (Euclidean distance for example) [111].

Recently, distributional similarity with context has appeared in leading computational linguisitics journals [40, 112, 133]. Their approaches, however, do not think about context in terms of a language (in the language-theoretic sense), but in terms of window-based context words or simple phrases.

In a XUTools-based approach, we compute document vectors for every string in a high-level language (such as sections or subsections). This approach allows us to apply machine-learning and information-retrieval techniques to a set of vectors that is in one-to-one correspondance with a high-level language. We argue that this could be a novel direction for thinking about distributional similarity in context where context is defined according to a language-theoretic notion of language. We define context in this manner so that *context* corresponds to high-level language constructs found in structured corpora.

Our initial work on this approach in the Classics attests to the validity of this claim. We applied this approach to some initial analyses of Homeric Scholia and published our results [117]. These results arguably changed 200-year old assumptions about the source of the scholia. This work used simple k-means and hierarchical clustering algorithms.

More recently, we used spectral clustering on document vectors that we extracted from a corpus of sections of IGTF PKI policies. The notion of *context* here, was defined as the language of sections across multiple versions of policies by the same CA.

Specifically, we chose policies from 5 different CAs. Each CA had between six and eight different versions of a policy and we analyzed the sections within those policies.

We computed an epsilon neighborhood graph as well as a disparity graph based on the similarity matrix computed between document vectors where a *document* corresponds to a section of policy. This initial work showed that there was indeed structure in the clusters that was worth exploring.

In addition, we might explore how the output of distributional similarity algorithms vary when we cluster document vectors of varying levels of abstraction. For example, we could explore and formalize how the clustering of IGTF PKI *sections* relates to the clustering of *subsections*.

8.3.5 Current XUTools:

We could think of `head` and `tail` as extensions of `xugrep` in which we only report structures within a given range of nodes within parse trees. Traditional `grep` has an `--invert-match` option, the analogue of which in `xugrep` would also be quite useful.

During our discussions with practitioners in the electrical power grid, many have expressed interest in the scalability of our tools. Both `grep` and `wc` are eminently parallel problems. In fact `grep` is an example used by the Apache Hadoop, a framework to run applications on large clusters of commodity hardware [61]. We could imagine distributing `xugrep` or `xuwc` according to subtrees of the xupath query tree. Furthermore, the Zhang and Shasha tree edit distance algorithm that we employ, has a parallel variant that we could use to implement a parallel `xudiff`.

8.3.6 New XUTools

Other practitioners have expressed the desire to be able to edit texts in terms of high-level structures. For example, a context-free `sed` or some other kind of structural editor. In fact, one practitioner has offered to help us design a xupath-based structural editor.[11] Before proceeding, we would need to survey the literature regard-

[11] Thanks to Mauricio Antunes.

ing structural editors.

Still, others have expressed the desire for a `sort` that operates on multiple levels of abstraction. In our earlier work we demonstrated how one could our `xugrep` combined with traditional `sort` to extract and sort data structures. A xusort, however, would sort data structures in place. One could imagine a radix-sort like functionality where instead of progressively sorting numbers according to place value, one could sort structures according to level of containment. The details of this, however, would need to be thought out more carefully.

8.3.7 A GUI for XUTools

Our interactions with power utilities and auditors have illustrated the necessity of a Graphical User Interface (GUI) for our XUTools. These practitioners do not want to be bothered with running command-line scripts, rather, they want a nice front end. We currently have two ideas for GUIs based upon XUTools.

Matlab for Textual Analysis: We have begun to explore the metaphor of Matlab for textual analysis. Just as researchers use Matlab in order to perform various numerical analyses, so could researchers use XUTools to perform textual analyses. We have begun to prototype this interface as a web application using Python CherryPy [26]. We demonstrated this visualization layer to the External Advisory Board (EAB) of our Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) project. Currently, we have a visual interface for the first auditing capability that we demonstrated to the TCIPG EAB. The visual interface allows auditors and administrators to pinpoint important security primitives within a power control network.

A Visual Structural Editor: Another GUI is a visual layer to the structural editor that we previously mentioned. The observation from which this visualization originates is that both forms and natural-language documents consist of field, value

pairs. In a form, fields have a type and the values are usually quite short. For example, an EMR may consist of a field for systolic and diastolic blood pressure, and the values of those fields may be validated against this type. In a natural-language document such as RFC 3647 or the outline for this thesis, we can think of each section and subsection within the text as a field, and the values as the contents of those passages.

By viewing form and natural-language document in this manner, we can allow practitioners to construct documents, not in terms of pages, but in terms of structure.

Our visual structural editor has applications to several problem scenarios because it reinvents word processing.[12] For brevity, we consider healthcare IT and operating systems.

First, the visual structural editor would address the gap between homogeneous EMRs and heterogenous workflows within the hospital. A hospital-wide EMR document schema could serve as the baseline for patient records, but individual clinics and doctors could modify this structure as needed and adapt content along a continuum of form-fielded data or unstructured text. For example, doctors in a clinic may decide to begin recording certain classes of information according to their own schema.

Second, our editor could provide a new interface into OS filesystems. OS filesystems currently rely upon directory structures for users to organize their information contained in files. A structural editor that allows people to author their own information hierarchies would enable people to arbitrarily organize their information at different levels of granularity and attach appropriate permissions to this data (if backed by the better memory trapping we discussed previously).

Finally, we could extend this visual editor with services based upon our libxutools. Just as we currently use word count, spell-check, or change tracking in traditional editors, so could we apply other services to process texts within our editor.

For example, we could provide a structural heat map that allows users to see where

[12]This statement is deliberately provocative.

and how much a document has changed as it was revised.[13] In one scenario, a legislator might need to quickly understand 11th-hour revisions to a bill. The legislator runs our tool over the document, and seconds later views the text as a sequence of sections with entries highlighted using a color gradient. For example, dark red highlighting might indicate more changes, while a light red color could indicate fewer changes. The legislator searches for all occurrences of a phrase in which she is interested and has results imposed over the sections. She sees that there is lots of change in a section with many search results and so *drills-down* to view a heatmap for the subsections.

8.4 Conclusions

In this chapter, we gave a preview of currently-ongoing research in the domains of X.509 PKI, terms of service policies, and the electrical power grid. In addition, we introduced many new possible avenues for XUTools as part of a broader research program to reconceive how people create, maintain, and analyze structured text. Finally, we discussed extensions to our existing codebase and theoretical toolbox to analyze structured text.

[13]According to Doug McIlroy, a similar visualization was provided for SCCS source code data at Bell Laboratory. The visualization, however, required manual annotation of the source code.

Chapter 9

Conclusions

Security policies define sets of rules that are designed to keep a system in a good state. Just as the words *system* and *good* can mean a variety of things in different domains, so can the term *security policy* have different meanings to practitioners.

During our fieldwork, we observed that security policies and artifacts that implement those policies come in a variety of different formats. Furthermore, these policy formats differed from Traditional Orange-Book formalizations that used lattices and matrices [76]. In practice, a security policy may be specified and implemented in a variety of formats that include natural-language legal documents, network device configurations, protocol messages, and system logs.

We observed that despite the variety of security policies that we encountered, many of these policies may be viewed as structured texts. We then related a variety of problems in a number of domains to three core limitations of security-policy analysis (1) the *Tools Gap Problem*, (2) the *Granularity of Reference Problem*, and (3) the *Policy Discovery Needs Problem.* (More information may be found in Chapter 2 which discusses our fieldwork, the security problems we encountered, and relates these problems to the three core limitations of security policy analysis.)

It appeared to us that we could help practitioners create and maintain policy

by formalizing security policy analysis with concepts from language theory, parsing, and discrete mathematics. We focused on formalisms that let us address our three core limitations of policy analysis. (More information on our theoretical toolbox for textual analysis may be found in Chapter 3.)

We then used these theoretical concepts to inform the design of XUTools. (More information about the design, implementation, and evaluation of XUTools may be found in Chapters 5 and 6.)

Our XUTools allow practitioners to process a broader class of languages than traditional `Unix` tools. Structured-text file formats transcend the limited syntactic capability of traditional Unix text-processing tools. We have designed and built XUTools, specifically `xugrep`, `xuwc`, and `xudiff`, to process texts in language-specific constructs.

Our applications of XUTools to network configuration management (Chapter 7), to enterprise security policies, and the elecrical power grid (Chapter 8) demonstrate that high-level language constructs may be used as units of measurement to measure properties of language constructs and quantify their evolution. Although our approach has a language-theoretic foundation, the practical implications of our tools means that practitioners can inventory, measure similarity, and see the usage of high-level language constructs in a variety of security policies. Finally, since many of these language constructs have names that persist across multiple versions of text, we can measure the evolution of security primitives through time.

Text is a sequence of characters and the early manuscripts reflected this. The earliest Ancient Greek sources did not even have punctuation. Over time, people started to identify meaningful substrings cand called them function blocks, headers, paragraphs, sentences, lines, pages, and books. We view these structures as languages in the language-theoretic sense; our XUTools process and analyze texts with respect to these languages just as people have been doing manually for thousands of years.

Appendix A

Pre-XUTools PKI Policy Analysis Tools

Before XUTools, we designed several other tools that helped practitioners by addressing the three core limitations of security policy analysis we introduced in Chapter 2. In this chapter, we will describe the design, implementation, and evaluation of these other tools.

Prior to XUTools, our research focused mainly on PKI policy analysis. During that time period, we prototyped several tools that addressed the gap between high-level language constructs used by policy analysts, allow practitioners to process texts on multiple levels of abstraction, and measure policy evolution. The sections of this chapter, describe the design, implementation, and evaluation of some of these other tools. We present the tools in the order in which they were originally developed.

A.1 PKI Policy Repository

We built the *PKI Policy Repository* so that we had access to a real-world dataset of versioned PKI Certificate Policies (CPs) and Certification Practices Statements (CPSs). Our repository, available at http://pkipolicy.appspot.com/ contains

roughly 200 versioned policies from the International Grid Trust Federation (IGTF).

The policies were converted from PDF into Text Encoding Initiative XML (TEI-XML) via our *policy-encoding toolchain*. We will describe this tool later in Section A.5. TEI-XML is a standard to represent texts in digital form [14]. Like previous efforts to encode policies using XML [18, 19], we model a security policy as a tree.[1] Given a policy's text, we mark up only its reference scheme, the outline of provisions defined in Section 6 of RFC 2527 or 3647 [27, 28]. This results in a policy representation that is both machine-actionable and human-readable.

A.1.1 Security Policy Analysis Problems Addressed

The PKI policy repository addresses the first two core limitations of security policy analysis.

First, our *PKI Policy Repository* bridges the gap between high-level languages used by analysts and the low-level languages upon which their tools operate. Figures A.1 and A.2 illustrate our point. In Chapter 2, we reported that although policy analysts operate on PKI policies in terms of their RFC 2527 or RFC 3647 reference structure, machine representations of policy such as PDF and Word documents are organized by page. This imposes a semantic gap that forces policy operations to be largely manual.[2] In contrast, our *PKI Policy Repository* allows analysts to retrieve passages of policy in terms of the RFC 2527 or RFC 3647 reference structure.

Second, our *PKI Policy Repository* allows practitioners to retrieve and thereby operate on security policies at a variety of levels of abstraction ranging from the entire document to individual sections, subsections, or even paragraphs of policy.

[1] Our approach was inspired by current approaches to digitize Classical texts [35, 116].

[2] Although PDFs may be built with bookmarks that are oriented to sections and subsections a gap still exists. See our discussion in Chapter 2.

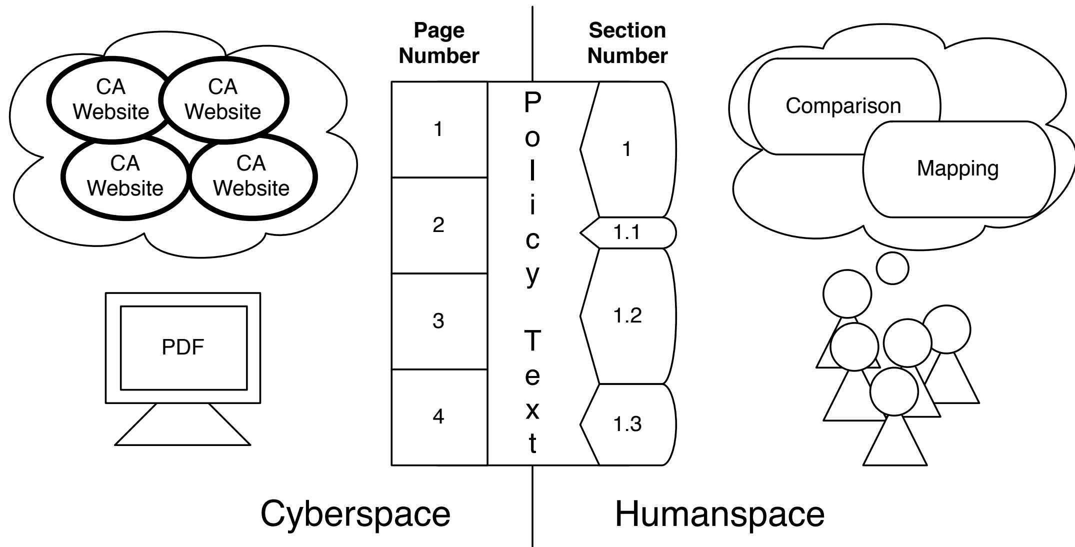


Figure A.1: Policy analysts operate on PKI policies by their reference structure, but machine representations of policy such as PDF and Word are written and operated on in terms of pages. This imposes a semantic gap and forces policy operations to be largely manual.

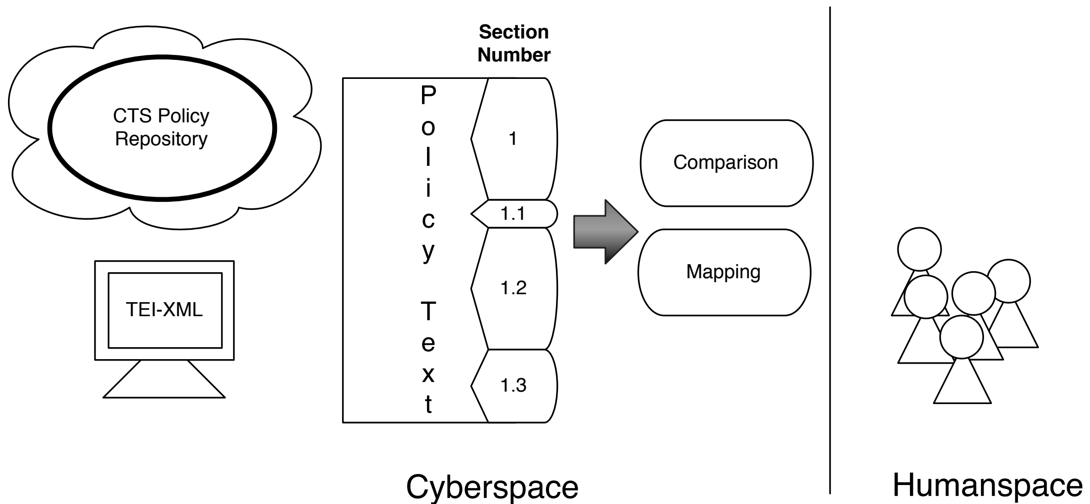


Figure A.2: Our representation of policy is machine-actionable but still human-readable as a legal document. Since our policies are machine actionable, we can encode analyses and thereby produce more reliable data for X.509 processes.

A.1.2 Design and Implementation

Our *PKI Policy Repository* is based on our prior work with Harvard's Center for Hellenic Studies and the College of the Holy Cross to develop the *Canonical Text Services (CTS)* protocol to reference and retrieve Homeric texts by their reference structure [116]. The CTS protocol uses HTTP to provide a simple REST-XML web service to retrieve canonically cited texts. Users or applications can retrieve sections of a policy by supplying a Canonical Text Services Uniform Resource Name (CTS-URN) and other HTTP request parameters.

Our *PKI Policy Repository* implements the CTS protocol using the Google AppEngine Python framework [54].

A.1.3 Evaluation

When we presented our PKI Policy Repository to practitioners at the FPKIPA, EUGridPMA, and TAGPMA, many analysts agreed that a policy repository was desirable to find and maintain policies and could streamline X.509 processes. Nonetheless, practitioners did express some concerns.

Some practitioners were concerned about the time it took to convert policies into our TEI-XML encoding. Certainly, they could see benefit once the policy was encoded, but the encoding process itself could be expensive. For example, to convert a PDF policy into TEI-XML, used to take us 4-6 hours of copying and pasting. Although some CA's already had an XML format for their policies, others were concerned about how long it would take to move a policy from PDF format into TEI-XML. In response to practitioners' concerns about excessive encoding time, we developed the *Policy Encoding Toolchain.*

In addition, practitioners were also concerned about the variation in section and subsection headings of PKI policies that roughly followed RFC 2527 or 3647 formats. In the real-world, headers might be relocated and paired with a different passage

reference. Analysts urged us to generalize our approach to handle header relocation. In response to practitioner concerns about variation in CP/CPS structure at the FPKIPA, we developed the *Vertical Variance Reporter* and eventually `xudiff` to detect the relocation of provisions.

We note that the data in our *PKI Policy Repository* may have value beyond the domain of identity management. Our repository is a corpus of versioned texts that discuss similar topics in slightly different language. The data is labeled by a standard reference scheme and might be useful for development of machine-learning algorithms for polysemy or for information retrieval algorithms that require a structured data source.[3]

A.2 Policy Builder

Our *Policy Builder* helps CAs to create new policies from a repository of extant, already accredited policies. In actual practice, a new certificate policy may be created when a CA wants to join a federation or bridge. CAs typically copy and paste passages of old policy into their new policy and selectively edit a few words and phrases as needed. The more similar the new, derivative certificate policy is to older, already accepted policies, the greater the chances for the new policy to be accepted. Under these circumstances, policy creation is quickly followed by policy review.

A.2.1 Security Policy Analysis Problems Addressed

Our *Policy Builder* attempts to address first two core limitations of security policy analysis. First, our *Policy Builder* closes the gap between machine-actionable content and the high-level langauges used by policy analysts by allowing practitioners to compose policy in terms of meaningful units of policy. Second, our *Policy Builder*

[3]Conversations with Dan Rockmore.

allows practitioners to import policy statements at a variety of levels of granularity ranging from sections, to subsections, to individual paragraphs if desired.

While Klobucar et al. have stated the need for machine-assisted policy creation [72], no tools have been built to fill this need and non have emerged that consider policy creation as a means to streamline policy review.

A.2.2 Design and Implementation

Our *Policy Builder* fills the need for machine-assisted policy creation while facilitating the review and evaluation of newly-created policies. Rather than copying and pasting policy statements from PDFs, as is the current CA practice, *Policy Builder* imports policy content directly from CPs in one or more *PKI Policy Repositories.* More specifically, the *Policy Builder* initializes an empty document template as defined in RFC 3647 and populates it with corresponding content from selected policies.

Policy content currently includes assertions, or security requirements qualified by MUST, SHOULD, or other adjectives from RFC 2119 that indicate significance [11]. Rather than copying and pasting content, policy assertions are imported into the new document by simply clicking on them. Once a document is built to satisfaction, the CA may serialize policy to XML, PDF, or HTML. Since each assertion includes a CTS-URN to its source policy, CAs can see how many security requirements they imported from bridge or grid-approved CPs. Similarly, reviewers may process the XML and filter original content from reused content.

A.2.3 Evaluation

Our *Policy Builder* received a fair amount of attention by practitioners. Most notably, at IDTrust 2010, a group from Motorola was interested in our Policy Builder as their group had thought of a similar tool but had not developed it as far as we had.[4]

[4]RIP Motorola.

A.3 Policy Mapper

Our *Policy Mapper* takes an RFC 2527 certificate policy and transforms it into an RFC 3647 structure using a mapping defined in Section 7 of RFC 3647 [28]. Originally, *Policy Mapper* was part of our *Policy Reporter* tool. Since policy analysts and CAs were most interested in our policy mapping feature, we will only discuss the *Policy Mapper* in this thesis. More information on our *Policy Reporter* may be found in our EuroPKI 2009 paper [140].

A.3.1 Security Policy Analysis Problems Addressed

Our *Policy Mapper* addresses our first core limitation of security policy analysis because it closes the gap between the language that practitioners use to analyze policy and the languages that practitioner tools can process.

A.3.2 Design and Implementation

Given a reference to a PKI policy, the *Policy Mapper* queries one or more *PKI Policy Repositories* to retrieve the policy in RFC 2527 format. Once the policy has been imported, the policy mapper instantiates a blank RFC 3647 policy and maps the RFC 2527 sections into the appropriate portions of the template.

A.3.3 Evaluation

To evaluate the *Policy Mapper*, we timed how long it took to automatically map the set of provisions of an RFC 2527 policy into the RFC 3647 structure. For this experiment we assumed that the policies being mapped were readily available on disk in PDF format for the manual case and were in the *PKI Policy Repository* as TEI-XML in the automated case. In addition, we used a highly-experienced certificate authority operator so that we could compare our approach to the fastest manual times

possible.[5]

We used the mapping defined in RFC 3647 and in three time trials the *Policy Mapper* completed the mapping in 50, 39, and 35 seconds respectively.

These results highlight the benefits of tools that can process the same high-level units as practitioners. In under one minute, provisions from one section of a certificate policy were automatically mapped. Our experienced CA estimated that mapping a policy from 2527 to 3647 format requires 20% more effort than a direct mapping between 3647 CPs. Considering that the average mapping takes 80-120 hours for an experienced CA, although the comparison is not exact, we claim that our results indicate a significant time savings in policy mapping. This claim was supported by practitioners at Protiviti and Digicert who repeatedly asked us to run our *Policy Mapper* on RFC 2527 documents.

We also want to note that in preparation for the experiment, automation of the mapping process immediately revealed an error in RFC 3647's mapping matrix: Section 2.1 in 2527 format maps to Section 2.6.4 in 3647 format. A closer look at RFC 3647, Section 6 revealed that Section 2.6.4 does not exist in the outline of provisions! Automatic mapping allows one to easily change a mapping and rerun the process as frequently as desired. Our approach also increases the transparency of the mapping process because generated RFC 3647 policies contain references to the source RFC 2527 provisions from which they are mapped. Finally, automatic policy mapping is easily reproduced; generated policies can be compared to other policies by loading them into the *PKI Policy Repository*. It took roughly 1 minute to load a policy into the repository depending upon the size of the CP/CPS.

[5]Thanks to Scott Rea.

A.4 Vertical Variance Reporter

Our *Vertical Variance Reporter* computed a mapping from the table of contents of one policy to the table of contents of another policy. The term *vertical variance* comes from Classical scholarship and describes how the reference structure of two versions of a text vary.

A.4.1 Security Policy Analysis Problems Addressed

We designed the *Vertical Variance Reporter* to address the practitioner community's concern over policy variation. After presenting our *PKI Policy Repository*, *Policy Builder*, and *Policy Mapper* to several IGTF PMAs and the FPKIPA, practitioners urged us to think about policy variation because in the real world, headers may be relocated, renamed, or paired with a different passage. Analysts encouraged us to generalize our approach to at least handle the relocation of headers.

A.4.2 Design and Implementation

Our *Vertical Variance Reporter* compares every provision in a *baseline* policy (that contains n provisions) to every provision in a policy under consideration (that contains m provisions) to produce an $n \times m$ matrix. Entry (i, j) in this matrix contains the value of the *Levenshtein distance metric* between the provision header i in the first policy and provision header j in the second policy.[6] Our tool processes the resultant matrix to report a mapping that classifies policy sections as matched, relocated, or unmapped.

[6]The *Levenshtein distance metric* is a string edit distance metric with insert, delete, and update operations.

A.4.3 Evaluation

To evaluate our *Vertical Variance Reporter*, we chose 10 policies from our *PKI Policy Repository*. Overall, we found that most policies in our test corpus followed the standard format described in RFC 2527 and 3647. The Federal Bridge Certificate Authority (FBCA) CP was an exception as it contained 28 non-standard provisions that were all below the subsection level. For example, Section 6.2.3.4 is found in FBCA CP but is not found in RFC 3647. If one considers only sections, subsections, and subsubsections, then we successfully identify between 97.8% and 100% of all actual provisions. We should note that these are our results from our final experiment and that more details are available in our 2010 IDTrust paper [141].

A.5 Policy Encoding Toolchain

We developed the *Policy Encoding Toolchain* to address the practitioner community's concern about the time it takes to convert a PKI policy to TEI-XML. During work on our EuroPKI 2009 paper, we found that we could convert a PDF policy to TEI-XML in 4-6 hours by copying and pasting content manually. Our *Policy Encoding Toolchain* reduces the time it takes to convert text-based PDF policies to our TEI-XML format.

We use three steps to encode a PDF policy with our *Policy Encoding Toolchain*. First we use Google Docs to generate Google's OCR HTML output for a given PDF policy. Second we parse this HTML to generate a TEI-XML encoding as well as CSS styling information. Finally, we generate a high-quality, human-readable view of the policy that faithfully recreates the typography seen in Google's OCR HTML.

We recently repurposed this keychain for our pilot studies of terms of service policies described in Chapter 8. We were able to adapt this tool to encode multiple versions of terms of service policies from a wide variety of service providers that included Apple, Facebook, and Comcast.

A.6 Conclusions

In this Chapter, we discussed the purpose, design, and evaluation of several non-XUTools tools that we developed to address the core limitations of PKI policy analysis. We note that our XUTools provides a more general framework to implement similar services. This chapter represents a subset of the most promising tools that we published in our EuroPKI 2009 and IDTrust 2010 papers [140,141]. For more details, please consult those sources.

Appendix B

PyParsing Internals

XUTools currently uses the PyParsing library [82]. The PyParsing library uses a *top-down parsing algorithm*, an algorithm that attempts to construct a parse tree for the input by starting at the root and building the tree in preorder [2].[1]

The PyParsing library uses higher-order functions called parser combinators to define the equivalent of a recursive-descent parser (with backtracking) for a context-free grammar. The functional programming term *higher-order function* refers to a function that takes one or more functions as input and outputs a function. *Parser combinators* are higher-order functions that take recognizers called interpretation functions as input and return another interpretation function.

In this chapter, we first provide background for recursive-descent parsers implemented via parser combinators. We then evaluate these parsers. Finally, we conclude with the implementation and evaluation of the `scan_string` method which is central to our `xugrep` algorithm.

[1] In contrast, a *bottom-up parsing algorithm* attempts to construct a parse tree for the input by starting at the leaves and building the tree in postorder.

B.1 PyParsing and Recursive Descent Parsers

We will now discuss how parser combinators may be used to construct the equivalent of a recursive-descent parser and specifically how the PyParsing library implements these concepts. We base our discussion on Wadler's presentation of the former topic [137]. Therefore, we will first define interpretation functions, and describe how PyParsing implements several of these functions. We will then define several parser combinators and describe their implementation in PyParsing. Finally, we will walk through an example (based on our corpus of PKI policies encoded in TEI-XML) that shows how parser combinators implement the equivalent of a recursive-descent parser when they process an input string.

B.1.1 Interpretation Functions

An *interpretation function* is a function that recognizes strings in a language. As shown in Figure B.1, given an input string w, an interpretation function f returns a list of results. If the input string is not in the language of the recognizer implemented by the interpretation function, then the list is empty. Otherwise, the list contains pairs of values derived from consumed input (such as tokens or a parse tree), and the remainder of unconsumed input (tail w). Each pair in the list represents a successful interpretation of w with respect to the language recognized by f. For example, the interpretation function for an ambiguous grammar may return a list of results with several interpretations.

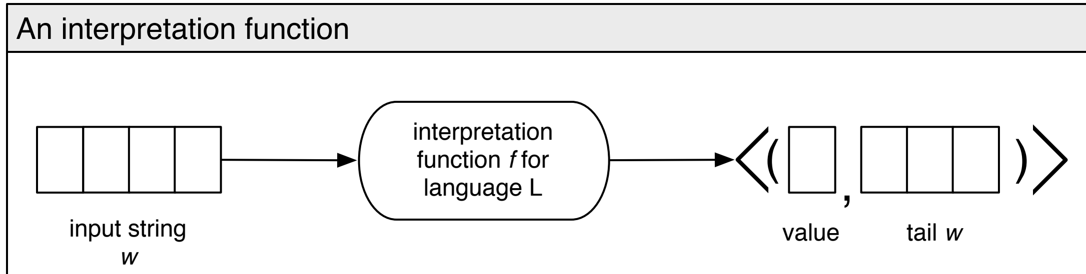


Figure B.1: An interpretation function implements a recognizer for language. Given an input string w it returns a list of results. If the list is empty, then there is no prefix of w that is in the language of the interpretation function. If the list is non-empty, then it contains a list of successful interpretations of w. Each interpretation consists of a value (a list of tokens or a parse tree derived from a prefix of w) and the tail of w that was unconsumed by f.

In the context of parsing, interpretation functions may be defined to recognize a variety of languages traditionally associated with lexical and syntactic analysis. Traditionally programs with structured input first divide the input into meaningful units (lexical analysis) and then discover the relationship among those units (syntactic analysis) [78]. We will discuss lexical analysis now and syntactic analysis during our discussion of parser combinators.

In traditional compiler construction, lexical analysis transforms the input sequence of characters into a sequence of tokens. A *token* is a name that references a set of strings that follow a rule called a *pattern* associated with the token. The pattern *matches* each string in the set. A *lexeme* is a character subsequence in the input stream that is matched by the pattern for the token [2].[2] Table B.1 provides examples of tokens, lexemes, and patterns for TEI-XML.

[2]This prose closely follows the pedagogy of Aho, Sethi, Ullman section 3.1.

Tokens for TEI-XML Grammar Fragment		
token	**sample lexemes**	**informal description of pattern**
div_start (end)	<tei:div>, <tei:div type="section">, <tei:div type="subsection">, <tei:div color="orange">, (</tei:div>)	start (end) of a division of text
head_start (end)	<tei:head> (</tei:head>)	start (end) of a passage header
paragraph_start (end)	<tei:p> (</tei:p>)	start (end) of paragraph
section_start	<tei:div type="section">	start of a section of text
subsection_start	<tei:div type="subsection">	start of a subsection of text

Table B.1: Each row in this table gives an example of a token, examples of strings in the language of the pattern associated with the token, and an informal description of the pattern. Lexical analyzers specify these patterns using regular expressions. Alternatively, we may write interpretation functions that recognize these languages.

Lexical analyzers specify token patterns using regular expressions but we may also write interpretation functions that recognize these languages. For example, in order to recognize the **div_start** token, we can write an interpretation function that looks for the "<" character, followed by the string "tei:div" and zero or more key/value attribute pairs, and terminated by the ">" character. In fact, PyParsing's `pyparsing.makeXMLTags` method constructs an interpretation function that does exactly what we just described.

PyParsing provides a number of additional methods and classes to construct interpretation functions that are traditionally associated with lexical analysis. The `pyparsing.Literal` class constructs an interpretation function to exactly match a specified string. The `pyparsing.Word` class constructs an interpretation function to match a word composed of allowed character sets (such as digits and alphanumerical characters). The `pyparsing.Keyword` class constructs an interpretation function to match a string as a language keyword. Finally the `pyparsing.Empty` class constructs an interpretation function that will always match as it implements the empty token.

B.1.2 Parser Combinators

A *parser combinator* is a high-order function that takes interpretation functions as input and returns another interpretation function. These combinator functions are often designed to model the form of a BNF grammar so that "the parser/interpreter explicitly expresses the grammar interpreted" [48]. For example, we will see that the *Or* combinator may be expressed as the | operator in PyParsing's syntax.

Parser combinators allow one to express relations among meaningful units of text recognized by interpretation functions. The discovery of relationships between meaningful units of text is the goal of traditional syntactic analysis—the meaningful units are tokens and the relations among those units are encoded by a parse tree.[3] The parse tree reflects the relationships among terminals (tokens) and nonterminals in a grammar. In the previous section we described how to construct interpretation functions for the former. We now discuss how to construct interpretation functions for the latter by means of parser combinators.

Although a variety of combinators may be defined, we will discuss the *Or* and *And* combinators because the syntax of these combinators reflects BNF form in PyParsing's grammar syntax.

Figure B.2 illustrates the behavior of the *Or* combinator [137]. Given an interpretation function f_1 for the language L_1 and an interpretation function f_2 for the language L_2, the *Or* combinator returns an interpretation function f_3 for the language $L_1 \cup L_2$.[4]

[3]Indeed, the edges of the parse tree are a *relation* in the mathematical sense. The ordering of child nodes within a parse tree is another important relation for syntactic analysis, however.

[4]We note that context-free languages are *closed* under the union operation—if L_1 and L_2 are both context-free, then their union $L_1 \cup L_2$ is context-free.

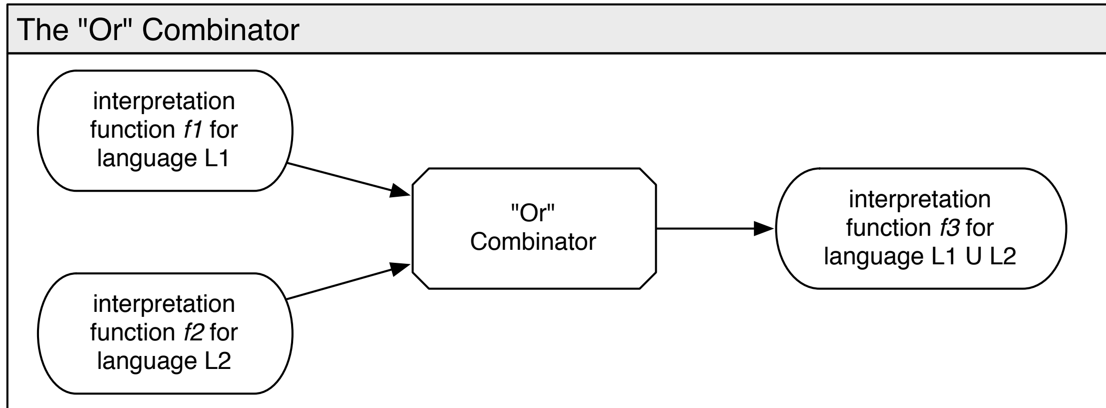


Figure B.2: *Or* combinator takes two interpretation functions f_1 and f_2 as input. Interpretation functions f_1 and f_2 recognize languages L_1 and L_2 respectively. The *Or* combinator constructs an interpretation function f_3 that recognizes the language $L_3 = L_1 \cup L_2$. The PyParsing syntax expresses the construction of f_3 as $f_1 | f_2$; this syntax is designed to resemble a BNF grammar.

In Figure B.3, we use the *Or* combinator to construct a recognizer for the language of subsection content (SUBSECTION_CONTENT) in PKI policies encoded with TEI-XML.

Figure B.5 illustrates the behavior of the *And* combinator [137]. This combinator may also be expressed as a binary operator on two interpretation functions f_1 and f_2 that recognize languages L_1 and L_2 respectively. The *And* combinator creates a recognizer, implemented as an interpretation function, that recognizes the language $L_1 \circ L_2$. In other words, it recognizes the language of strings that consist of a string prefix in L_1 followed by a substring in L_2.[5]

[5] Again, we note that the context-free languages are closed under the concatenation operation. If L_1 and L_2 are both context-free, then their concatenation $L_1 \circ L_2$ is also context-free.

```
PyParsing Grammar Fragment for TEI-XML

# token definitions
div_start, div_end = makeXMLTags("tei:div")
head_start, head_end = makeXMLTags("tei:head")
paragraph_start, paragraph_end = makeXMLTags("tei:p")
subsection_start  = copy(div_start).setParseAction( withAttribute( type="subsection" ) )
section_start = copy(div_start).setParseAction( withAttribute( type="section" ) )

# CFG for TEI-XML
HEAD_CONTENT = SkipTo(MatchFirst(head_end))
HEAD = head_start + HEAD_CONTENT + head_end

PARAGRAPH_CONTENT = SkipTo(MatchFirst(paragraph_end))
PARAGRAPH = paragraph_start + PARAGRAPH_CONTENT + paragraph_end

DIV = Forward()
DIV_CONTENT = HEAD | PARAGRAPH | DIV
DIV = nestedExpr( div_start, div_end, content = DIV_CONTENT )

SUBSECTION_CONTENT = HEAD | PARAGRAPH | DIV
SUBSECTION = subsection_start + ZeroOrMore( SUBSECTION_CONTENT ) + div_end

SECTION_CONTENT = HEAD | PARAGRAPH | DIV
SECTION = section_start + ZeroOrMore( SECTION_CONTENT ) + div_end
```

Figure B.3: The PyParsing API uses parser combinators so that input tokenization and syntactic analysis may be expressed using code that resembles a traditional grammar. This figure shows a grammar fragment for the dialect of TEI-XML Sections in which our X.509 TEI-XML policies are encoded. Productions that correspond to tokens traditionally recognized during lexical analysis are in bold. Figure B.4 presents a portion of policy in the language of this grammar.

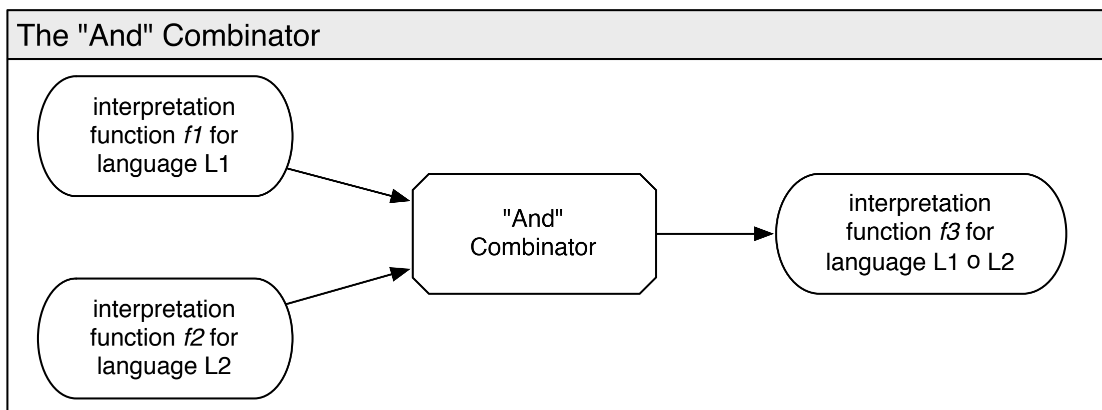


Figure B.5: The *And* combinator may be viewed as a binary operator + that recognizes strings in the language of the concatenation of its two input recognizers f_1 and f_2. Specifically, given two interpretation functions f_1 and f_2 as input, the *And* combinator constructs an interpretation function f_3 that recognizes the language $L_3 = L_1 \circ L_2$.

Sample TEI-XML PKI CPS Section (Based on IGTF DFN-GridGermany CPS version 1.4)	
human-readable	**TEI-XML encoding**
6 TECHNICAL SECURITY CONTROLS The requirements for technical security measures of a CA or RA are determined by the types of services offered. The precise level of security… **6.1 KEY PAIR GENERATION AND INSTALLATION** **6.1.1 KEY PAIR GENERATION** Key pairs for the Grid-CA are generated on a dedicated IT system unequipped with networking capability or directly within a Hardware Security Module (HSM). **6.1.1.1 HSM REQUIREMENTS** The keys are stored only on external data storage media and are protected by a PIN or when generated within a HSM the keys are protected by the HSM. **6.1.2 PRIVATE KEY DELIVERY TO SUBSCRIBER** No cryptographic key pairs are generated for subscribers **…**	<tei:div type="section" n="6"> **<tei:head>TECHNICAL SECURITY CONTROLS</tei:head>** <tei:p>The requirements for technical security measures of a CA or RA are determined by the types of services offered. The precise level of security ...</tei:p> <tei:div type="subsection" n="1"> **<tei:head>KEY PAIR GENERATION AND INSTALLATION</tei:head>** <tei:div type="subsubsection" n="1"> **<tei:head>KEY PAIR GENERATION</tei:head>** <tei:p>Key pairs for the Grid-CA are generated on a dedicated IT system unequipped with networking capability or directly within a Hardware Security Module (HSM).</tei:p> <tei:div type="subsubsubsection" n="1"> **<tei:head>HSM REQUIREMENTS</tei:head>** <tei:p>The keys are stored only on external data storage media and are protected by a PIN or when generated within a HSM the keys are protected by the HSM.</tei:p> </tei:div> </tei:div> <tei:div type="subsubsection" n="2"> **<tei:head>PRIVATE KEY DELIVERY TO SUBSCRIBER</tei:head>** <tei:p>No cryptographic key pairs are generated for subscribers.</tei:p> </tei:div> **…** </tei:div> </tei:div>

Figure B.4: This figure shows a portion of a Certification Practices Statement (CPS) and its encoding in TEI-XML, an XML dialect that we used to encode natural-language security policies. The policy shown on the left consists of one section (Technical Security Controls), one subsection (Key Pair Generation and Installation), two subsubsections (Key Pair Generation and Private Key Delivery to Subscriber) and one subsubsubsection (HSM Requirements). On the right, we see each of the corresponding sections encoded as nested `tei:div` elements and the headers are highlighted in bold for readability. The portion of policy shown is in the language of the grammar in Figure B.3.

In Figure B.3, we use the *And* combinator to construct a recognizer for the language of sections encoded with TEI-XML. We use the PyParsing syntax to specify an interpretation function for subsections and this syntax resembles a grammar production in BNF form. According to this definition, a SECTION consists of a subsection start tag (**section_start**), followed by zero or more content strings, followed by an end tag (**div_end**). We use the *And* combinator to construct an interpretation function for this language.

B.1.3 Combinators and Recursive-Descent Parsers

As mentioned earlier, we can use parser combinators to define the equivalent of a recursive-descent parser (with backtracking) for a context-free grammar. We just explained how the *And* and *Or* combinators may be used to construct an interpretation function for sections encoded in TEI-XML. The syntax for this construction closely aligns with the syntax for a production in a BNF grammar. When we call the interpretation function for a production on an input string w, the resultant call graph aligns with the series of calls made by a recursive-descent parser.

As mentioned earlier, recursive-descent parsers attempt to construct a parse tree for an input string w by starting at the root and building the parse tree in preorder. Recall from Chapter 3 that an interior node v of a parse tree is labeled by a nonterminal and the children of v are ordered, from left to right, by symbols in the right side of the nonterminal's production that was applied to derive the tree.

Figure B.6 illustrates how the HEAD production in Figure B.3 specifies an interpreter function that, when called on an input w, implements a recursive-descent parser. A step-by-step description of the call sequence is described in Table B.2.

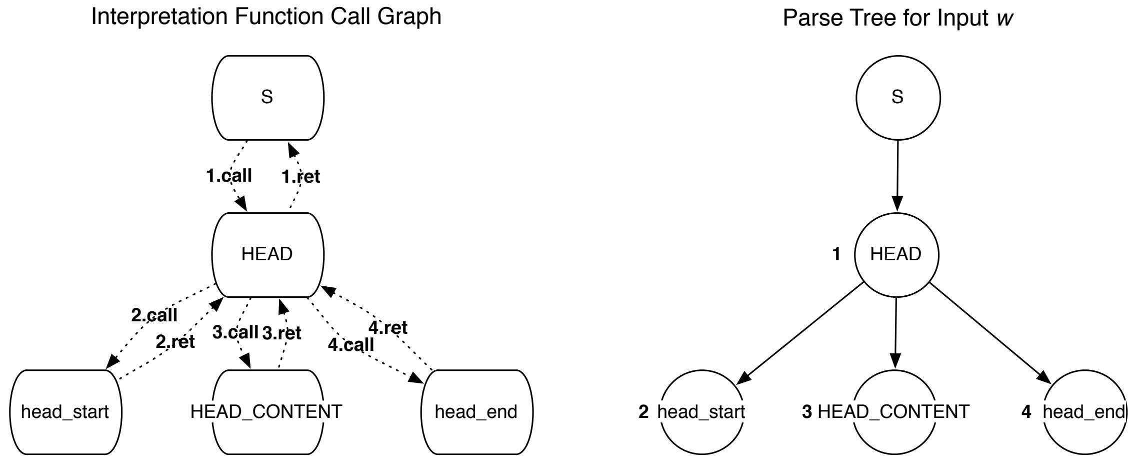


Figure B.6: When we call an interpretation function constructed using parser combinators, the resultant call graph implements a recursive-descent parser. Here, we see that the parse tree for input w is built in preorder. Furthermore, the *stack* for this recognizer is implicitly represented as the activation stack for the interpretation function calls shown in the *Step* column in Table B.2. The step-by-step description of this process is available in Table B.2.

Step	Call Graph Description	Parse Tree Description
0	Call the interpretation function for the start production S. We augment the grammar with a start symbol so that the start production is guaranteed not to refer to itself.	Add a child node to the parse tree root.
1.call	Call the interpretation function for the first (and only) HEAD production with input *w*.	Add a child node to the parse tree root. Set the language name field of the child node to "HEAD".
2.call	Call the interpretation function for the head_start language with input *w*.	Add a child node to the parse tree root. Set the language name field of the child node to "head_start".
2.return	The interpretation function for the head_start language returns [("<tei:head>", "TECHNICAL…</tei:head>")].	Set the value of node 2's text field to "tei:head".
3.call	Call the interpretation function for HEAD_CONTENT language with input *w2* = "TECHNICAL…</tei:head>".	Add a second child node to the parse tree root. Set the language name field of the child node to "head_content".
3.return	The interpretation function for HEAD_CONTENT language returns [("TECHNICAL…CONTROLS", "</tei:head>")].	Set the value of node 3's text field to "TECHNICAL…CONTROLS".
4.call	Call interpretation function for head_end language with input *w3* = "</tei:head>".	Add a third child node to the parse tree root. Set the language name field of the child node to "head_end".
4.return	The interpretation function for the head_end language returns [("</tei:head>", [])].	Set the value of node 4's text field to "</tei:head>".
1.return	The interpretation function for HEAD returns a list of the first element in the tuples returned from calls 2-4. [(["<tei:head>", "TECHNICAL…CONTROLS", "</tei:head>"], [])].	Set the value of node 1's text field to "<tei:head>TECHNICAL…CONTROLS</tei:head>".

Table B.2: The entries in this table show how the interpretation function for the *head* production, specified in Figure B.3, implements a recursive-descent parser when invoked on an input. We note that we augment the grammar with a start symbol so that the start production does not refer to itself, thus keeping the grammar in Chomsky normal form. For more information about Chomsky normal form, consult [115].

B.2 Evaluation of Recursive Descent Parsers

We now evaluate our PyParsing recursive-descent parsers in terms of implementation complexity, recognition power, usability and portability, and against other parsing algorithms.

B.2.1 Evaluation—Implementation Complexity

First, we discuss the time and space complexity of our recursive-descent parser implemented via PyParsing.

Time Complexity: In the worst case, a recursive-descent parser may require backtracking. *Backtracking* occurs when the parser must make repeated scans of the input. For example, consider Figures B.8 and B.9. In this example, we have modified the grammar of Figure B.3 to include an alternative encoding of TEI header content by adding another production (HEAD_CONTENT2). Figure B.7 shows the modifications to the grammar.

Figure B.8 shows that when we apply our recursive-descent parser to the input string w, the parser tries the first HEAD production listed, and successfully matches the opening header tag (**head_start**). The parser then applies the HEAD_CONTENT2 production and the interpretation function for the word TECHNICAL succeeds. Unfortunately, the interpretation function for the word CONTROLS fails because the unconsumed input string contains the word SECURITY. Therefore, the HEAD_CONTENT2 production also fails as does the first HEAD production.

PyParsing Grammar Fragment for TEI-XML Headers

```
HEAD_CONTENT2 = Word("TECHNICAL") + Word("CONTROLS")
HEAD_CONTENT = SkipTo(MatchFirst(head_end))
HEAD = head_start + HEAD_CONTENT_2 + head_end
         | head_start + HEAD_CONTENT + head_end
```

Figure B.7: We modified the grammar in Figure B.3 to include an alternative encoding of TEI header content. We use this grammar in Figures B.8 and B.9 to illustrate backtracking.

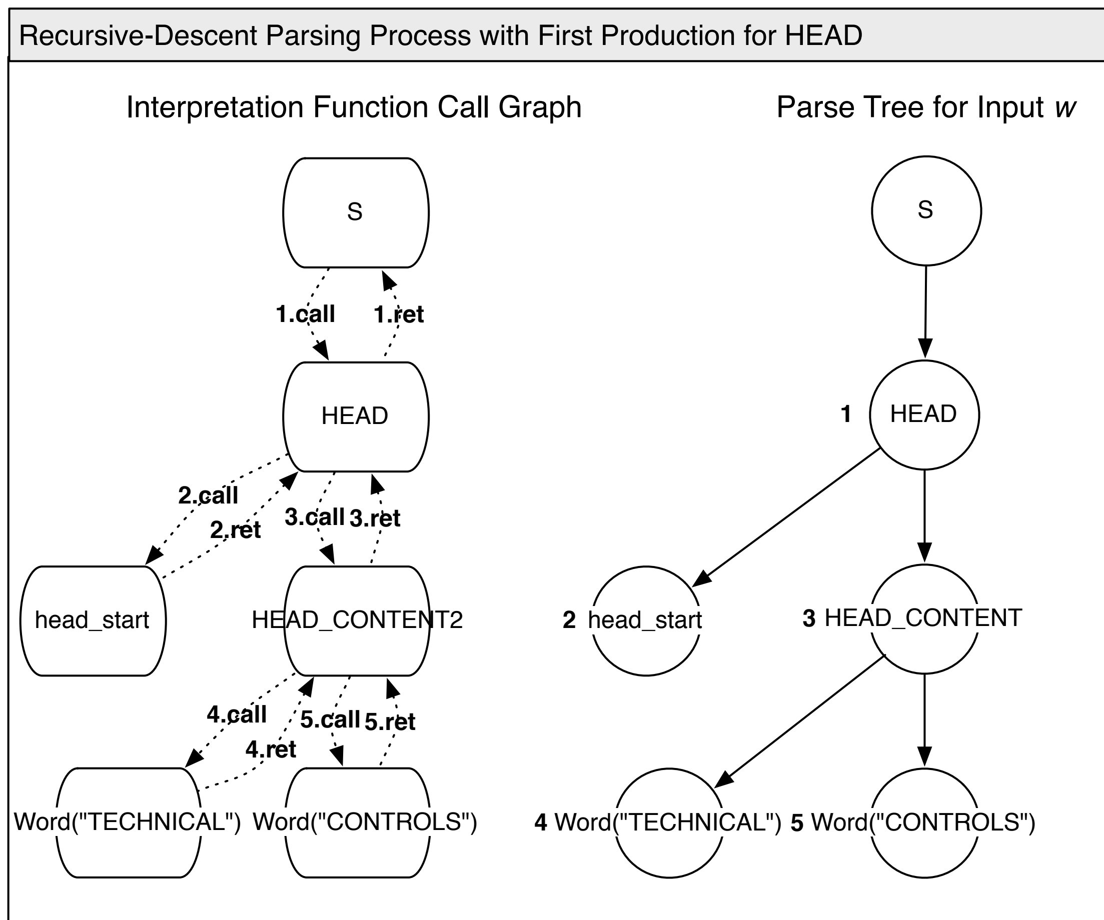


Figure B.8: Given an input string w, a recursive descent parser may repeatedly scan the same input string as it tries alternative productions for a nonterminal. This figure shows a recursive-descent parser that must backtrack after the interpretation function for the word CONTROLS fails. Therefore, the parser must backtrack. Repeated calls to the same interpretation function with the same input string make recursive-descent parsers with backtracking run in exponential time. In this example, the **head_start** interpretation function is called twice on the input string w.

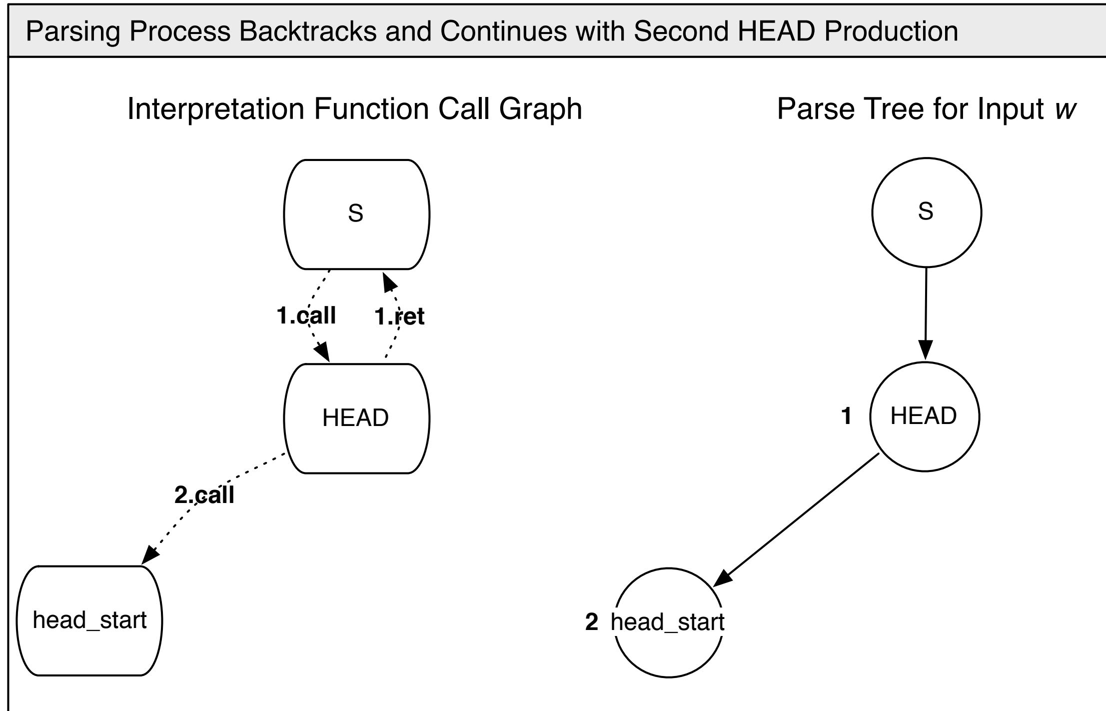


Figure B.9: The parsing process from Figure B.8 continues with the next HEAD production. Repeated calls to the same interpretation function with the same input string make recursive-descent parsers with backtracking run in exponential time. In this example, the **head_start** interpretation function is called twice on the input string w.

Figure B.9 shows that the parsing process backtracks and continues with the second HEAD production. A parser that must backtrack requires exponential time in the number of symbols of input in the worst case. This can occur because of repeated calls to the same interpretation function given the same input string. For example, in Figures B.8 and B.9 we can see that the **head_start** interpretation function is called twice on the input string w, once for every execution of HEAD. (Specifically, this can take $O(k^n)$ time for an LL(k) grammar and an input of length n.) In order to address this exponential time, a *packrat parser* uses memoization to store the results of invoking an interpretation function on an input string [44]. A packrat parser runs in $O(n)$ time given an input string of length n. The PyParsing library enables packrat parsing via the `pyparsing.enablePackrat` method. We note that the packrat parser resolves ambiguous text in favor of the first alternative, and thus

changes the interpretation of the grammar.

Space Complexity: A packrat parser requires linear space in the worst case (O(n)) to store the results of an interpretation function on an input string w that is n symbolss long. The results of an interpretation function are uniquely determined by the prefix it consumes. If the input string is length n, then there are $n + 1$ possible suffixes used in the recursive calls to a production's interpretation function (including w and the empty string). Therefore, since the number of interpretation functions is constant, and there are $n + 1$ possible results per interpretation function, the packrat parser takes $O(n)$ space to store intermediate parsing results [44].

B.2.2 Evaluation—Recognition Power

Recursive-descent parsers only operate on LL(k) grammars. An *LL grammar* is a grammar that can be parsed by scanning the input from left to right (the first L), and by constructing a leftmost derivation (the second L).[6] Finally, the k means that the parser needs to only look at most k symbols ahead to determine which production to apply [2]. An LL(k) grammar may not be ambiguous, nor can it have left-recursion. A left-recursive grammar can cause a recursive-descent parser to enter an infinite loop.

Nonetheless, Frost et al. recently showed that one can construct parser combinators for ambiguous, left-recursive grammars. The usability of such grammars we leave to potential future work [47].

B.2.3 Evaluation—Usability and Portability

PyParsing's specification of recursive-descent parsers allows one to construct a parser in a syntax that closely resembles defining a grammar. In general, recursive-descent parsers are easy to implement because they reflect closely the structure of a grammar.

[6] A parser computes a *leftmost derivation* if during the parsing process, the parser applies a production to the leftmost nonterminal in the current derivation.

This makes for very readable parser code and for an open-source project like XUTools this makes our parser interface more accessible to people that are interested in working on the project's source.

On the other hand, the PyParsing syntax is still an API, and we currently do not distinguish between the specification of a grammar and the construction of a parser for the language of that grammar. As a result, our "grammars" contain PyParsing-specific methods and this reduces the overall portability of our grammars. Since our grammar specifications are currently tied to the PyParsing library, we cannot use the grammars that we have written with a future, non-Python implementation of XUTools.

B.2.4 Alternative Parsing Algorithms

Although parser combinators allow us to easily construct recursive-descent parsers that use linear time and space if memoization is enabled, we may want to consider bottom-up parsing algorithms such as LR parsers. We leave this as potential future work, but discuss a few points on LR parsers here.

Recognition Power and Complexity: LR(k) parsers scan the input from left to right (the L), and construct a rightmost derivation (the R) by scanning at most k input symbols ahead. Aho, Sethi, and Ullman state that typical language constructs can usually be expressed using grammar that can be recognized with an LR parser [2]. An LR(k) parser may not be ambiguous, but it can be left-recursive.

Even though LR(1) parsers are difficult to implement by hand, parser-generators such Yacc [148] can generate a LR(1) parser directly from a grammar using the Lookahead-LR (LALR) construction. In addition, techniques such as association rules accommodate ambiguity within an LR parser even though an ambiguous grammar is not an LR(1) grammar [2].

Usability and Portability: Although LR parsers may be generated from a grammar, writing an LR grammar to generate such a parser can be difficult. In order to interpret parser error messages, the grammar writer must understand the LALR algorithm and this is less intuitive than understanding recursive-descent parsing. In fact, the GCC C++ parser was changed from LALR to recursive-descent parsers because the LALR error messages were difficult to interpret [50, 75].

If we were to use an LALR parser, then we could not expect end users to be able to write small grammars to use with XUTools. In contrast, the relation between recursive-descent parsers and a grammar specification is more easily understood with parser combinators. Furthermore, packrat parsers run in $O(n)$ time and consume $O(n)$ space and, according to Ford [44], packrat parsers can recognize any language by an LL(k) or LR(k) grammar.

B.3 Implementation and Evaluation of Scan String

In Chapter 5 we mentioned the `scan_string` method. The `scan_string` method takes string as input and a `language_name` for a production in a context-free grammar and outputs all non-overlapping substrings that are in that language.

Currently, we implement this method with the `pyparsing.scan_string` method. We note that the `pyparsing.scan_string` method does report overlapping matches if the `overlap` flag is set.

The `pyparsing.scan_string` method operates on a string of characters (as opposed to tokens) because PyParsing interpretation functions parse language constructs that are traditionally viewed as tokens as well as higher-level constructs reserved for syntactic analysis.

Given an input string w and an interpretation function for a language construct, the `pyparsing.scan_string` method initializes a location variable (`loc`) to 0. The

location variable holds the index for the start of the input string w' for the interpretation function.

Recall from the previous subsection that an interpretation function takes the input w' and tries to construct a derivation for a prefix of w'. The interpretation function returns a list of pairs that represent possible interpretations of w' and each pair contains a value (it could be a derivation) along with the tail of w' that was unconsumed by the interpretation function. This is an important point because traditional parsers take an input w' (where symbols are tokens, not characters) and try to construct a derivation for w' *in its entirety.* If a traditional parser cannot construct a derivation for all of w' then, the parse fails.

Since interpretation functions can construct derivations for prefixes of an input string w', the `scan_string` method need only keep track of the index for the start of w'. The `scan_string` method repeatedly calls the appropriate interpretation function on input strings w'. If w' has a prefix in the language of the interpretation function, then the `loc` counter is incremented by the length of the consumed input. Otherwise the `loc` counter is incremented by 1. Either way, a string w', starting at index `loc` in w, is ready to be interpreted. In the worst case, the method invokes the interpretation function once for every character in w and constructs $O(n)$ strings as input to the interpretation function. As discussed above, the interpretation function is equivalent to a recursive-descent parser if packrat mode is turned off. In this case, `scan_string` takes exponential time. If packrat mode is turned on, then the interpretation function takes time linear in the input and so `scan_string` interprets $O(n)$ strings, and it takes $O(n)$ time per interpretation. Therefore, with packrat mode, `scan_string` takes $O(n^2)$ time.

Furthermore, each string that `scan_string` gives to an interpretation function as input is at most n characters long. If I assume that packrat mode reuses the same memory space every time that it invokes the interpretation function f, then

the space complexity of `scan_string` is linear since there are a constant number of interpretation functions that f may invoke and there are at most $n+1$ possible results per invoked interpretation function.

Comparision Against Scan String with Traditional Parsers

We now describe the worst-case analysis of a `scan_string` method with a traditional parser. As mentioned in the previous passage, traditional parsers take an input string and try to construct a derivation. A traditional parser fails if the input string is not in the language of the parser's grammar.

To scan a string w using a traditional parsing algorithm, we need to consider the beginning and the end of substrings of w' because a traxditional parser finds derivations if and only if w' is in the language of the parser's grammar.

Therefore, a scan string algorithm that uses a traditional parser must index the start and end positions of w' within the input string w. There are $O(n^2)$ such substrings in a string of length n. Therefore, in the worst case, a scan string algorithm can take cubic time in the length of the input w (in tokens) when the parsing algorithm takes linear time. Again, if we assume that the parsing algorithm reuses space between invocations, then the space complexity of the scan string algorithm with traditional parsers is the space complexity of the traditional parser.

B.4 Conclusions

In this chapter, we gave a detailed description of a how we use PyParsing's parser combinators to implement a recursive-descent parser for XUTools. We then concluded that although other parsing algorithms may be available, parser-combinators provide a usable interface for users of our XUTools to specify their own parsers for small, lightweight grammars.

Bibliography

[1] 4BF - Four Bridges Forum, 2009. Retrieved May 29, 2009 from http://www.the4bf.com/.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1988.

[3] Apel, S., Liebig, J., Lengauer, C., Kastner, C., and Cook, W. R. Semistructured merge in revision control systems. In *Proceedings of the Fourth International Workshop on Variability Modeling of Software Intensive Systems (VaMoS 2010)* (January 2010), University of Duisburg-Essen, pp. 13–20.

[4] Arnold, G. NIST Smart Grid Program Overview, March 2012. Retrieved November 28, 2012 from http://www.nist.gov/smartgrid/upload/Smart_Grid_Program_Review_overview_-_arnold_-_draft1.pdf.

[5] Ball, E., Chadwick, D., and Basden, A. The implementation of a system for evaluating trust in a PKI environment. *Trust in the Network Economy, Evolaris 2*, unknown (unknown 2003), 263–279.

[6] Barnes, K., and Johnson, B. National SCADA test bed substation automation evaluation report. Tech. Rep. 15321, Idaho National Laboratory (INL), INL Critical Infrastructure Protection/Resiliance Center, Idaho Falls, Idaho 83415, October 2009.

[7] Batz, D., Brenton, J., Dunn, D., Williams, G., Clark, P., Elwart, S., Goff, E., Harrell, B., Hawk, C., Henrie, M., Kenchington, H., Maughan, D., Kaiser, L., and Norton, D. Roadmap to achieve energy delivery systems cybersecurity. Tech. rep., Department of Homeland Security, Cyber Security R&D Center, 333 Ravenswood Avenue, Menlo Park, CA 94025, September 2011.

[8] Benson, T., Akella, A., and Maltz, D. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (April 2009), USENIX Association, pp. 335–348.

[9] Bille, P. A survey on tree edit distance and related problems. *Theoretical Computer Science 337*, unknown (June 2005), unknown.

[10] Blaze, M., Feigenbaum, J., and Lacy, J. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (SP 1996)* (May 1996), IEEE, pp. 164–173.

[11] Bradner, S. IETF RFC 2119: Key words for use in RFCs to indicate requirement levels, March 1997.

[12] Bratus, S., Locasto, M. E., Patterson, M. L., Sassaman, L., and Shubina, A. Exploit programming: From buffer overflows to weird machines and theory of computation. *USENIX ;login:* (December 2011), 13–21.

[13] Bright, P. Amazon's lengthy cloud outage shows the danger of complexity. *ArsTechnica* (April 2012). Retrieved September 19, 2012 from http://goo.gl/z2nPq.

[14] Burnard, L., and Bauman, S. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, 5 ed., 2007.

[15] Extended Validation SSL Certificates—The Certification Authority/Browser Forum, 2012. Retrieved December 3, 2012 from https://www.cabforum.org.

[16] Caldwell, D., Lee, S., and Mandelbaum, Y. Adaptive parsing of router configuration languages. In *Internet Network Management Workshop, 2008. (INM 2008)* (October 2008), IEEE, pp. 1–6.

[17] Cantor, S., Kemp, J., Philpott, R., and Maler, E. Assertions and protocols for the OASIS Security Assertion Markup Language (SAML), 2005.

[18] Casola, V., Mazzeo, A., Mazzocca, N., and Rak, M. An innovative policy-based cross certification methodology for public key infrastructures. In *Proceedings of the 2nd European PKI Workshop (EuroPKI 2005)* (June and July 2005), EuroPKI, pp. 100–117.

[19] Casola, V., Mazzeo, A., Mazzocca, N., and Vittorini, V. Policy formalization to combine separate systems into larger connected network of trust. In *Proceedings of the IFIP TC6 / WG6.2 & WG6.7 Conference on Network Control and Engineering for QoS, Security and Mobility (Net-Con 2002)* (October 2002), IFIP, pp. 425–411.

[20] Catteddu, D., and Hogben, G. Cloud computing: Benefits, risks, and recommendations for information security, 2009.

[21] Chadwick, D., and Otenko, A. RBAC policies in XML for X.509 based privilege management. In *Proceedings of IFIP TC11 17th International Conference On Information Security (SEC' 2002)* (May 2002), Kluwer Academic, pp. 39–53.

[22] SunView Software, ChangeGear. Retrieved February 3, 2012 from http://www.sunviewsoftware.com/.

[23] Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)* (June 1996), ACM, pp. 493–504.

[24] Chen, K., Schach, S. R., Yu, L., Offutt, J., and Heller, G. Z. Open-source change logs. *Empirical Software Engineering 9*, 3 (September 2004), 197–210.

[25] Chen, X., Mao, Z. M., and Van der Merwe, J. Towards automated network management: Network operations using dynamic views. In *Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management (INM '07)* (August 2007), ACM, pp. 242–247.

[26] CherryPy – A Minimalist Python Web Framework, 2011. Retrieved November 28, 2012 from http://www.cherrypy.org.

[27] Chokhani, S., and Ford, W. IETF RFC 2527: Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework, March 1999.

[28] Chokhani, S., Ford, W., Sabett, R., Merrill, C., and Wu, S. IETF RFC 3647: Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework, November 2003.

[29] Cisco IOS configuration fundamentals command reference. Retrieved September 19, 2012 from http://www.cisco.com/en/US/docs/ios/12_1/configfun/command/reference/frd1001.html.

[30] Cobéna, G., Abiteboul, S., and Marian, A. Detecting changes in XML documents. In *Proceedings of the 18th International Conference on Data Engineering* (February and March 2002), IEEE, pp. 41–52.

[31] Coccinelle: A program matching and transformation tool for systems code, 2011. Retrieved November 11, 2011 from http://coccinelle.lip6.fr/.

[32] Codeanalyzer. Retrieved May 17, 2012 from http://sourceforge.net/projects/codeanalyze-gpl/.

[33] Common Information Model (CIM)/ energy management. Tech. Rep. 61970-1, International Electrotechnical Commission IEC, December 2007. Available on November 23, 2012 from http://webstore.iec.ch/webstore/webstore.nsf/ArtNum_PK/35316.

[34] Configuring IP Access Lists, December 2007. Retrieved December 7, 2012 from http://www.cisco.com/en/US/products/sw/secursw/ps1018/products_tech_note09186a00800a5b9a.shtml.

[35] Crane, G. The Perseus digital library, 2009. Retrieved May 29, 2009 from http://www.perseus.tufts.edu/hopper/.

[36] Criticism of Facebook. *Wikipedia* (2012). Retrieved November 28, 2012 from http://en.wikipedia.org/wiki/Criticism_of_Facebook.

[37] DAY-CON: Dayton Security Summit, 2012. Retrieved November 28, 2012 from http://day-con.org/.

[38] Designs, S. Semantic designs: Smart differencer tool. Retrieved May 16, 2012 from http://www.semdesigns.com/Products/SmartDifferencer/.

[39] *diff(1) Manual Page*, September 1993. Retrieved May 17, 2012.

[40] Dinu, G., and Lapata, M. Measuring distributional similarity in context. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing (EMNLP 2010)* (Stroudsburg, PA, USA, 2010), Association for Computational Linguistics, pp. 1162–1172.

[41] Energy Services Provider Interface (ESPI). Tech. Rep. REQ.21, North American Energy Standards Board (NAESB), 2010. Retrieved November 23, 2012 from http://www.naesb.org/ESPI_Standards.asp.

[42] FINLEY, K. Putting an end to the biggest lie on the internet. *TechCrunch* (August 2012). Retrieved November 29, 2012 from http://techcrunch.com/2012/08/13/putting-an-end-to-the-biggest-lie-on-the-internet/.

[43] FLURI, B., WURSCH, M., AND GALL, H. C. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE '07)* (October 2007), IEEE Computer Society, pp. 70–79.

[44] FORD, B. Packrat parsing: A practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology (MIT), 32 Vassar Street, Cambridge MA 02139, 2002.

[45] FOTI, N., HUGHES, J. M., AND ROCKMORE, D. N. Nonparametric sparsification of complex multiscale networks. *PLoS ONE 6* (February 2011).

[46] IDManagement.gov—Criteria and Methodology, 2012. Retrieved December 5, 2012 from http://idmanagement.gov/pages.cfm/page/Criteria-and-Methodology.

[47] FROST, R., HAFIZ, R., AND CALLAGHAN, P. Parser combinators for ambiguous left-recursive grammars. *Practical Aspects of Declarative Languages* (2008), 167–181.

[48] FROST, R., AND LAUNCHBURY, J. Constructing natural language interpreters in a lazy functional language. *The Computer Journal 32*, 2 (1989), 108–121.

[49] Garfinkel, S., Nelson, A. J., and Young, J. A general strategy for differential forensic analysis. In *Proceedings of the 12th Conference on Digital Research Forensics (DFRWS '12)* (August 2012), ACM, pp. 550–559.

[50] GCC 3.4 Release Series—Changes, New Features, and Fixes—GNU Project—Free Software Foundation (FSF), 2012. Retrieved December 9, 2012 from http://gcc.gnu.org/gcc-3.4/changes.html.

[51] Geer, D. Monoculture on the back of the envelope. *USENIX login; 30*, 6 (2005), 6–8.

[52] Gold, B. WEBTrust / Client FAQ, 1997-2004. Retrieved May 29, 2009 from http://www.webtrust.net/faq-client.shtml.

[53] Google privacy changes in break of EU law. *BBC News* (March 2012). Retrieved November 29, 2012 from http://www.bbc.co.uk/news/technology-17205754.

[54] Google appengine, 2011. Retrieved November 4, 2011 from http://code.google.com/appengine/.

[55] Apache Wave - Welcome to Apache Wave (Incubating), 2012. Retrieved November 28, 2012 from http://incubator.apache.org/wave/.

[56] *grep(1) Manual Page*, January 2002. Retrieved May 17, 2012.

[57] Grimm, R., and Hetschold, T. Security policies in OSI-management experiences from the DeTeBerkom project BMSec. In *Proceedings of JENC6: Bringing the World to the Desktop* (May 1995), unknown, p. unknown.

[58] Group, T. S. G. I. P. C. S. W. Guidelines for smart grid cyber security. Tech. Rep. 7628, National Institute of Standards and Technology (NIST), 100 Bureau Drive, Stop 1070, Gaithersburg, MD 20899, August 2010.

[59] Grunschlag, Z. cfgrep - context free grammar egrep variant, 2011. Retrieved November 11, 2011 from http://www.cs.columbia.edu/~zeph/software/cfgrep/.

[60] Grushinskiy, M. XMLStarlet command line XML toolkit, 2002. Retrieved May 15, 2012 from http://xmlstar.sourceforge.net/.

[61] Welcome to Apache Hadoop!, 2012. Retrieved November 28, 2012 from http://hadoop.apache.org/.

[62] Harris, C. IT downtime costs $26.5 billion in lost revenue. *InformationWeek* (May 2011). Retrieved November 23, 2012 from http://www.informationweek.com/storage/disaster-recovery/it-downtime-costs-265-billion-in-lost-re/229625441.

[63] Housley, R., and Polk, T. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*. Wiley Computing Publishing, New York, NY, 2001.

[64] Substation automation. Tech. Rep. 61850-1, International Electrotechnical Commission IEC, April 2003. Available on November 23, 2012 from http://webstore.iec.ch/webstore/webstore.nsf/ArtNum_PK/30525.

[65] Inglesant, P., Sasse, M. A., Chadwick, D., and Shi, L. L. Expressions of expertness: The virtuous circle of natural language for access control policy specification. In *Proceedings of the 4th Symposium on Usable Privacy and Security (SOUPS '08)* (July 2008), ACM, pp. 77–88.

[66] ISO/IEC 27001:2005 information technology – security techniques – information security management systems – requirements, 2008.

[67] Israeli, A., and Feitelson, D. G. The linux kernel as a case study in software evolution. *Journal of Systems Software 83*, 3 (March 2010), 485–501.

[68] Jaakkola, J., and Kilpelainen, P. Using sgrep for querying structured text files. In *Proceedings of SGML Finland 1996* (November 1996).

[69] Jensen, J. Presentation for the CAOPS-IGTF session, March 2009.

[70] Kay, M. XSL transformations (XSLT) version 2.0, 2002.

[71] Kim, H., Benson, T., Akella, A., and Feamster, N. The evolution of network configuration: A tale of two campuses. In *Proceedings of the Internet Measurement Conference (IMC 2011)* (November 2011), ACM, pp. 499–512.

[72] Klobučar, T., and Jerman-Blažič, B. A formalisation and evaluation of certificate policies. *Computer Communications 22*, 12 (July 1999), 1104–1110.

[73] Koorn, R., van Walsem, P., and Lundin, M. Auditing and certification of a public key infrastructure. *ICASA Journal 5* (2002). http://www.isaca.org/Journal/Past-Issues/2002/Volume-5/Pages/Auditing-and-Certification-of-a-Public-Key-Infrastructure.aspx.

[74] Koppel, R., and Gordon, S. *First, Do Less Harm: Confronting the Inconvenient Problems of Patient Safety*. ILR Press, 2012.

[75] LALR Parser. *Wikipedia* (2012). Retrieved December 9, 2012 from http://en.wikipedia.org/wiki/LALR_parser.

[76] Latham, D. *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, 1986.

[77] Lehman, M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE 68*, 9 (September 1980), 1060–1076.

[78] Levine, J. R., Mason, T., and Brown, D. *lex & yacc*. O'Reilly, Cambridge, Massachusetts, 1992.

[79] Lim, Y. T., Cheng, P.-C., Rohatgi, P., and Clark, J. A. Dynamic security policy learning. In *Proceedings of the 1st ACM Workshop on Information Security Governance (WISG '09)* (November 2009), ACM, pp. 39–48.

[80] Lutterkort, D. Augeas—a configuration API. In *Proceedings of the Linux Symposium* (July 2008), pp. 47–56.

[81] McGuire, P. Subset C parser (BNF taken from the 1996 international obfuscated C code contest). Retrieved May 16, 2012 from http://pyparsing.wikispaces.com/file/view/oc.py.

[82] McGuire, P. pyparsing, 2012. Retrieved September 19, 2012 from http://pyparsing.wikispaces.com.

[83] Mendes, S., and Huitema, C. A new approach to the X.509 framework: Allowing a global authentication infrastructure without a global trust model. In *Proceedings of the 1st Symposium on Network and Distributed System Security (NDSS '95)* (February 1995), ISOC, pp. 172–189.

[84] Metrics 1.3.6. Retrieved May 17, 2012 from http://metrics.sourceforge.net/.

[85] Moore, W. H. National transportation statistics. Tech. rep., U.S. Department of Transportation, April 2012. Section B, Table 11-1. Retrieved December 3, 2012 from http://www.bts.gov/publications/national_transportation_statistics/html/table_01_11.html.

[86] eXtensible Access Control Markup Language (XACML), 2005.

[87] Greatest engineering achievements of the 20th century, 2003. Retrieved November 23, 2012 from http://www.greatachievements.org/.

[88] NERC CIP reliability standards, 2012. Retrieved November 11, 2011 from http://www.nerc.com/page.php?cid=2|20.

[89] NetAPT Access Policy Tool, 2012. Retrieved September 19, 2012 from https://www.perform.csl.illinois.edu/netapt/index.html.

[90] O'Callaghan, D. Automated Certificate Checks, January 2009. Presented at the 15th EUGrid PMA.

[91] Open Source Tripwire. Retrieved February 3, 2012 from http://sourceforge.net/projects/tripwire/.

[92] Oppenheimer, D., Ganapathi, A., and Patterson, D. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)* (March 2003), USENIX Association, p. unknown.

[93] Information Warfare Simulation — PacketWars, 2012. Retrieved November 28, 2012 from http://packetwars.com/.

[94] Pala, M., Cholia, S., Rea, S. A., and Smith, S. W. Extending PKI interoperability in computational grids. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '08)* (May 2008), IEEE Computer Society, pp. 645–650.

[95] Parr, T. ANTLR parser generator. Retrieved May 17, 2012 from http://www.antlr.org/.

[96] Pauley, W. A. Cloud provider transparency: An empirical evaluation. *IEEE Security and Privacy 8*, 6 (Nov.–Dec. 2010), 32–39.

[97] Pike, R. Structural regular expressions. Retrieved December 10, 2012 from http://doc.cat-v.org/bell_labs/structural_regexps/.

[98] Plonka, D., and Tack, A. J. An analysis of network configuration artifacts. In *The 23rd Conference on Large Installation System Administration (LISA '09)* (November 2009), USENIX Association.

[99] 1003.1-2008—IEEE Standard for Information Technology—Portable Operating System Interface (posix(r)). Retrieved December 7, 2012 from http://standards.ieee.org/findstds/standard/1003.1-2008.html.

[100] Windows PowerShell. Retrieved February 3, 2012 from http://technet.microsoft.com/en-us/library/bb978526.aspx.

[101] Press, A. Lee MacPhail dies at 95 in Florida. *ESPN MLB* (November 2012). Retrieved December 3, 2012 from http://espn.go.com/mlb/story/_/id/8611020/lee-macphail-hall-famer-ex-al-president-dies-95.

[102] Public power annual directory & statistical report. Tech. rep., American Public Power Association (APPA), May 2012. Retrieved November 23, 2012 from http://www.publicpower.org/files/PDFs/USElectricUtilityIndustryStatistics.pdf.

[103] RANCID - Really Awesone New Cisco Config Differ, 2010. Retrieved December 1, 2010 from http://www.shrubbery.net/rancid/.

[104] Rea, S. DigiCert Management Bios: Scott Rea. Retrieved December 3, 2012 from http://www.digicert.com/news/bios-scott-rea.htm.

[105] RELAX NG home page. Retrieved May 17, 2012 from http://www.relaxng.org/.

[106] BMC Remedy IT Service Management. Retrieved February 3, 2012 from http://www.bmc.com/solutions/itsm/it-service-management.html.

[107] Researchers expanding diff, grep Unix Tools, 2011. Retrieved May 17, 2012 from http://tech.slashdot.org/story/11/12/08/185217/researchers-expanding-diff-grep-unix-tools.

[108] Rosen, K. H. *Discrete Mathematics and Its Applications.* McGraw-Hill College, 2006.

[109] Routray, R., and Nadgowda, S. CIMDIFF: Advanced difference tracking tool for CIM compliant devices. In *Proceedings of the 23rd Conference on Large Installation System Administration (LISA '09)* (October and November 2009), USENIX Association, p. unknown.

[110] Rubin, P., and MacKenzie, D. *wc(1) Manual Page*, October 2004. Retrieved May 16, 2012.

[111] Salton, G., Wong, A., and Yang, C.-S. A vector-space model for automatic indexing. *Communications of the ACM 18*, 11 (1975), 613–620.

[112] Séaghdha, D. O., and Korhonen, A. Probabilistic models of similarity in syntactic context. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2011)* (Stroudsburg, PA, USA, 2011), Association for Computational Linguistics, pp. 1047–1057.

[113] Shubina, A. Accessibility and Security or: How I Learned to Stop Worrying and Love the Halting Problem. In *Proceedings of the 2012 Dartmouth Computer Science Research Symposium (CSRS 2012)* (September 2012).

[114] Sinclair, S., Smith, S. W., Trudeau, S., Johnson, M., and Portera, A. Information risk in financial institutions: Field study and research roadmap.

In *Proceedings of 3rd International Workshop on Enterprise Applications and Services in the Finance Industry (FinanceCom2007)* (December 2008), pp. 165–180.

[115] Sipser, M. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, Massachusetts, 2006.

[116] Smith, D. N. Canonical Text Services (CTS), 2009. Retrieved May 29, 2009 from http://cts3.sourceforge.net/.

[117] Smith, D. N., and Weaver, G. A. Applying domain knowledge from structured citation formats to text and data mining: Examples using the CITE architecture. In *Proceedings of Text Mining Services TMS Conference* (March 2009), Common Language Resources and Technology Infrastructure (CLARIN), p. unknown.

[118] Smith, S. W. *Trusted Computing Platforms*. Springer, 2005.

[119] Smith, S. W., and Koppel, R. Healthcare information technology's relativity problems: A typology of how mental models, reality and HIT differ. *Submitted for Publication* (2012).

[120] Smith, S. W., and Marchesini, J. *The Craft of System Security*. Addison-Wesley, Boston, Massachusetts, 2008.

[121] Operational Intelligence, Log Management, Application Management, Enterprise Security and Compliance — Splunk, 2012. Retrieved November 28, 2012 from http://www.splunk.com/.

[122] Change Management—Network Change Control and IT Configuration Management Process — Splunk. Retrieved November 28, 2012 from http://www.splunk.com/view/change-monitoring/SP-CAAACP2.

[123] Staff, P. Justices to lawyers: Don't make us read the law. *Politico* (March 2012). Retrieved November 28, 2012 from http://www.politico.com/news/stories/0312/74601.html.

[124] State of New Hampshire Residential Care and Health Facility Licensing Patients' Bill of Rights Section 151:21. Retrieved November 28, 2012 from http://www.dartmouth.edu/~health/docs/PatientBillOfRights.pdf.

[125] Sun, X., Sung, Y. W., Krothapalli, S., and Rao, S. A systematic approach for evolving VLAN designs. In *Proceedings of the 29th IEEE Conference on Computer Communications (INFOCOM 2010)* (March 2010), IEEE Computer Society, pp. 1–9.

[126] Sung, Y.-w. E., Rao, S., Sen, S., and Leggett, S. Extracting network-wide correlated changes from longitudinal configuration data. In *Proceedings of the 10th Passive and Active Measurement Conference (PAM 2009)* (April 2009), unknown, pp. 111–121.

[127] Tekli, J., Chbeir, R., and Yetongnon, K. An overview on XML similarity: Background, current trends and future directions. *Computer Science Review 3*, 3 (August 2009), 151–173.

[128] Thomas, O. Almost every website on the planet could be in legal jeopardy, thanks to zappos. *Business Insider* (October 2012). Retrieved December 7, 2012 from http://www.businessinsider.com/zappos-terms-of-service-ruled-invalid-2012-10.

[129] TkDiff. Retrieved February 3, 2012 from http://tkdiff.sourceforge.net/.

[130] TOSBack — The Terms-Of-Service Tracker, 2012. Retrieved Sunday, February 12, 2012 from http://www.tosback.org/timeline.php.

[131] The IT Security Conference—Troopers, 2012. Retrieved December 3, 2012 from http://www.troopers.de.

[132] Trček, D., Jerman-Blažič, B., and Pavešić, N. Security policy space definition and structuring. *Computer Standards & Interfaces 18*, 2 (March 1996), 191–195.

[133] Van de Cruys, T., Poibeau, T., and Korhonen, A. Latent vector weighting for word meaning in context. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (2011), EMNLP '11, Association for Computational Linguistics, pp. 1012–1022.

[134] Vanbrabant, B., Joris, P., and Wouter, J. Integrated management of network and security devices in it infrastructures. In *The 25th Conference on Large Installation System Administration (LISA '11)* (December 2011), USENIX Association, p. unknown.

[135] Vil. Retrieved May 17, 2012 from http://www.1bot.com/.

[136] Von Luxburg, U. A tutorial on spectral clustering. *Statistics and Computing 17*, 4 (2007), 395–416.

[137] Wadler, P. How to Replace Failure by a List of Successes. In *Proceedings of A Conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1985), Springer-Verlag New York, Inc., pp. 113–128.

[138] Walsh, N., and Muellner, L. DocBook: The definitive guide, July 1999.

[139] Weaver, G. A., Foti, N., Bratus, S., Rockmore, D., and Smith, S. W. Using hierarchical change mining to manage network security policy evolution. In *Proceedings of the 11th USENIX Conference on Hot Topics in*

Management of Internet, Cloud, and Enterprise Networks and Services (HotICE 2011) (March–April 2011), USENIX Association, p. unknown.

[140] Weaver, G. A., Rea, S., and Smith, S. A computational framework for certificate policy operations. In *Proceedings of the 6th European PKI Workshop (EuroPKI 2009)* (September 2009), EuroPKI, pp. 17–33.

[141] Weaver, G. A., Rea, S., and Smith, S. W. Computational techniques for increasing PKI policy comprehension by human analysts. In *Proceedings of the 9th Symposium on Identity and Trust on the Internet (IDTrust 2010)* (April 2010), Internet 2, pp. 51–62.

[142] Weaver, G. A., and Smith, S. W. Context-free grep and hierarchical diff (poster). In *Proceedings of the 25th Large Installation System Administration Conference (LISA '11)* (December 2011), USENIX Association, p. unknown.

[143] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ASPLOS-X, ACM, pp. 304–316.

[144] Wolf, J. OMG WTF PDF: What you didn't know about Acrobat. In *Proceedings of the 27th Chaos Communication Congress (27C3)* (December 2010).

[145] xmllint. Retrieved May 16, 2012 from http://xmlsoft.org/xmllint.html.

[146] W3C XML Schema. Retrieved May 17, 2012 from http://www.w3.org/XML/Schema.

[147] XML Path language (XPath), 1999. Retrieved May 15, 2012 from http://www.w3.org/TR/xpath/.

[148] The LEX and YACC Page. Retrieved May 17, 2012 from http://dinosaur.compilertools.net/.

[149] Zazueta, J. Micro XPath grammar translation into ANTLR. Retrieved May 16, 2012 from http://www.antlr.org/grammar/1210113624040/MicroXPath.g.

[150] Zhang, K. *The editing distance between trees: algorithms and applications.* PhD thesis, New York University (NYU), New York, USA, 1989.

[151] Zhang, K., and Shasha, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Jounal of Computing 18*, 6 (December 1989), 1245–1262.