9-17-2011

BGrep and BDiff: UNIX Tools for High-Level Languages

Gabriel A. Weaver
Dartmouth College

Sean W. Smith
Dartmouth College

BGrep and BDiff: UNIX Tools for High-Level Languages (Extended Abstract)[1]

Dartmouth Computer Science Technical Report TR2011-705

Version of September 17, 2011

Gabriel A. Weaver and Sean W. Smith[2]

Dartmouth College

Abstract

The rise in high-level languages for system administrators requires us to rethink traditional UNIX tools designed for these older data formats. We propose new *block*-oriented tools, *bgrep* and *bdiff*, operating on *syntactic blocks* of code rather than the line, the traditional information container of UNIX. Transcending the line number allows us to introduce *longitudinal diff*, a mode of *bdiff* that lets us track changes across arbitrary blocks of code. We present a detailed implementation roadmap and evaluation framework for the full version of this paper. In addition we demonstrate how the design of our tools already addresses several real-world problems faced by network administrators to maintain security policy.

Keywords: UNIX, configuration management, security policy

1 Introduction

High-level languages for system administrators are increasingly on the rise over flat-file or line-based formats. Traditional UNIX tools, however, are geared towards these old file paradigms. In light of the variety of modern programming languages available to system and network administrators, and given the success and utility of *grep* and *diff*, we adapt and extend *grep* and *diff* for this new programming ecosystem.

Our new tools operate on *syntactic blocks* of code and text *as the default information container*, in contrast to traditional tools that treat the *line* as the default. Syntactic blocks often correspond to meaningful higher-level constructs ranging from a network interface in Cisco IOS, to a `VirtualHost` in Apache, to a section of text within a normative reference document such as an IETF RFC.

Our block-based *grep* gives adminstrators a general mechanism to extract blocks of text or code that match simple context-free patterns. Similarly, our block-based *diff* empowers administrators to compare two versions of the same set of blocks over time. Moreover, we provide a natural extension to *diff*, *longitudinal diff*, that enables an administrator to track changes made to a specific set of blocks (such as network interfaces) as a time-series dataset.

High-Level Languages for System Administration are on the Rise System administrators are increasingly turning to high-level programming languages to configure and manage their systems. Consider system configuration in general. Last summer's *USENIX Configuration Management Summit* featured four tools available to system administrators to programmatically configure their systems: CFEngine3, Bcfg2, Chef, and Puppet [1]. CFEngine3 explicitly encodes promises among different system resources. Bcfg2 views configuration management "as an API" for programming a system configuration. Chef views infrastructure as code that can benefit from software engineering practices. Finally, the Puppet language models the desired state of datacenters.

We also notice this trend in network configuration management. At *USENIX HotICE* this year, one Cisco engineer remarked that Cisco IOS is old and that model-driven architectures are the new direction to configure networks. Consider Netconf, Yang, and Cisco NxOS. In addition, some consider network configuration a form of distributed programming.

Traditional UNIX Tools were Designed for Simpler File Formats Two traditional UNIX workhorses are grep and diff. However, these tools were designed for simpler file formats than modern system modeling languages. During conversations with Doug McIlroy (inventor of *diff* and *pipes*, and arguably the first user of the first UNIX), we identified several design decisions behind grep and their limitations [10].

[1]This work was supported in part by the NSF (under grant CNS-0448499), and the TCIPG project sponsored by the DOE (under grant DE-OE0000097). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of any of the sponsors.

[2]{gweave01,sws}@cs.dartmouth.edu

Grep First consider *grep*. Users can casually and quickly write regular expressions to extract structure from a file. Originally, *grep* was written because Doug McIlroy needed to extract terms from a dictionary but the dictionary was too large to load in his text editor. Therefore, the regular expression parser was written in order to accommodate the limitations of the machine at the time.

Today, however, many of the languages and language constructs encountered by administrators are not regular. There are a wide variety of languages found within configuration management packages such as Chef. The unit of distribution within Chef, the *cookbook*, includes (among other things) configuration *templates* written in a configuration language specific to that utility, and *recipes*, Ruby scripts that specify how to manage node resources.

Given the variety of languages, we cannot assume they are all regular. Although Ruby supports blocks of code that are nested arbitrarily deep, one cannot write a regular expression that matches blocks of code at an arbitrary depth [13].

We want to be able to extract syntactic blocks because they often encode meaningful units of information. A practitioner may want to extract a particular method or set of methods from a Ruby recipie to see how Powershell is installed depending upon the version of Windows on hosting node. Alternatively, a practitioner may want to extract the set of virtual hosts to quickly see how she has partitioned their organizational domain into meaningful subdomains on a single server instance.

Diff *Diff* lets system administrators compare files by line. Today *diff* is a backbone in many version-control systems. More recently, *diff* is used by the *Really Awesome New Cisco Config Differ (RANCID)*, to report changes to the configuration of a network device.

Many of the languages encountered by system administrators today, however, are no longer organized by line number but in more complex syntactic structures. For example, Cisco IOS uses blocks of code to denote interfaces. Today, line numbers are as much a consequence of data storage as they are of language syntax. We argue that using the line as the primary information unit in these high-level languages is like trying to compare two editions of a textbook by page number rather than by logical section.

Practitioners agree with the need to reconsider line-based *diff* given today's file formats. Bcfg2 lets administrators compare configuration *models* rather than files. These models include *bundles*, a logical grouping on packages, files, and other entities that describe services on managed machines.

Bcfg2 represents configuration *models* as XML and its administrator comparison script (*bcfg2-admin compare*) walks this XML tree between two versions in order to report changes.

However, XML comparison is not a general solution to the variety of languages encountered by the system or network administrator including Cisco IOS. One network administrator at Dartmouth Computing Services expressed interest in being able to compare arbitrary blocks of configuration files such as network interfaces, and generate reports of how address-groups and other high-level constructs are used within those interfaces over time.

This Paper In Section 2 of this paper, we relate the requirements for a *diff* and *grep* in the modern sysadmin ecosystem to well-understood problems within theoretical computer science. In Section 3 we present a roadmap for the implementation and evaluation of our tools *bdiff* and *bgrep*. Section 4 orients our research in terms of hierarchical and longitudinal change, especially in network configuration management. Finally Section 5 concludes.

2 Rethinking Diff and Grep

In this section, we describe two adaptations of *grep* and *diff* to the variety of data seen by system and network administrators today. Having motivated the need to rethink these tools, we now recast these tools' requirements in terms of long-studied, theoretical problems from computer science.

Our redesign of grep should empower administrators to casually extract "lightweight" context-free structure just as readily as they currently extract regular structure using traditional *grep*.[3] Our redesign of diff

[3]We note that the term "lightweight" comes from Miller's Ph.D thesis [14].

should enable administrators to compare "parse-trees" of context-free structure just as readily as they can compare lines using traditional *diff*.

2.1 A Grep for "Lightweight" Grammars

We rethink *grep* in terms of the languages used by modern system and network administrators and map these requirements to the problem of defining lightweight grammars and parsing a file given a context-free (or sensitive) structure.

Modern Requirements First, let us rethink *grep* in terms of how system administration and the computing environment in general has changed since *grep*'s arrival in March 3, 1973. We focus on the increase in high-level, model-driven configuration languages.

One consequence of the increase in model-driven configuration languages is that meaningful constructs of interest to administrators are represented as nested blocks of code that span multiple lines. Examples include the aformentioned `VirtualHost` construct in Apache2, and the `interface` of Cisco IOS.

In contrast, the traditional UNIX environment of the 1970s used the line as a default information container. For example, the original *join* command was designed to join on the entire line and only used the first field as a key. However, as tabular formats became more common, the ability to parse and join on arbitrary fields contained within each line was added. Consider *awk*, which operates upon *records* which are default identified with lines.

Parsing and Lightweight Grammars Just as Thompson adapted the original *join* command to fielded data, so do we propose to adapt *grep* to context-free structures that are increasingly prevalent due to the rise in model-driven configuration.

Traditionally, writing a grammar and generating a parser for a language has not been viewed as a casual, spur-of-the-moment activity (as writing a regexp for *grep* might be). Although a great deal of time is involved in writing a grammar for an *entire* language, we explore the idea of writing a grammar for a subset of the entire language. For example, a system admin interested in `VirtualHost`s across multiple machines does not need a parser for Apache's *entire* configuration language, but just `VirtualHost` blocks. Likewise, a network admin interested in her network `interface`s does not need a parser for Cisco IOS, but a parser for `interface` blocks (despite the fact that Cisco IOS lacks a formal grammar).

Consequently, we need something more. Such a *grep* for context-free or sensitive structures would require a system admin to specify a grammar for the structures of interest and a set of files from which to extract matching structures. We believe this can be done by generating a parser for the context-free language that ignores all but the desired structure and parsing the set of files. In Section 3 we describe the implementation of our *bgrep* in more depth.

We should note that the process of identifying and understanding the generation of parse trees for subsets of a given language may help avoid differential parse tree attacks [15].

2.2 A Diff for Parse Trees

We now rethink *diff* in terms of these lightweight, context-free structures used throughout modern configuration languages to model systems and networks. We map the requirements of such a difference engine to the *Hierarchical Change Detection Problem* described by Chawathe et al. [6].

Modern Requirements Let us rethink *diff* in terms of the modern ecosystem of a system administrator. Certainly, many things have changed since 1976 when the original *diff* paper was published. Among these many things include the increase in high-level, model-driven configuration languages and the increase in cheap storage.

The increase in model-driven languages translates into an increased need to compare blocks of code across multiple editions of the same notional file. For example, an administrator of Apache2 may want

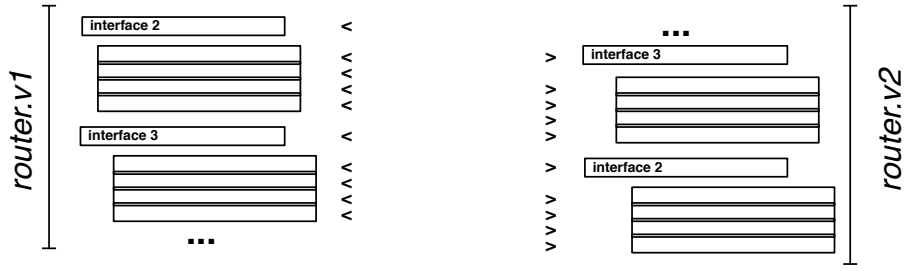

Figure 1: Traditional line-based diff is too low-level because it ignores the block-based structure defined by Cisco IOS. This real-world example from our research motivates the need for a block-based diff. The ellipses indicate multiple blocks of code before and after the blocks depicted that caused longest common subsequence (the algorithm used by *diff*) to fail.

to compare two configurations of a particular `VirtualHost`. Alternatively, a network administrator may want to compare two configurations of a particular `interface`.

As a consequence of the increase in cheap storage, we now have the luxury of reconsidering the window of time over which system admins monitor changes to a configuration. Just as system administrators use Nagios to monitor their infrastructure, we propose a *longitudinal diff* extension so that system admins can monitor changes within a configuration as time-series data.

In contrast, *longitudinal diff* wasn't a readily-available option for storage-limited, general-purpose computers of the 1970s. Revision-control systems such as SCCS *had* to store the deltas between versions of files rather than the files themselves due to storage limitations. Tools like RANCID hint at looking at changes longitudinally, but being based on line diffing, do not enable admins to track changes to arbitrary sections of a configuration *model* over time.

Hierarchical Change Detection We propose to adapt *diff* to context-free structures by leveraging Chawathe's work on the *Hierarchical Change Detection* problem [6]. Our *bgrep* (block-based grep) returns a forest of parse trees that match a given structure. If we apply our *grep* to two versions of the same file, then we can compare the two versions of each of these parse trees.

The question then becomes, how do we compare these two versions of the same notional block? We could simply recover the text by visiting the nodes of the parse tree in a prefix order and then compare the text. However, depending upon the programming language we would be eliminating some useful information. For example, in Cisco IOS, permuting two lines in an `interface` block has no effect on router functionality but shows up as a change in RANCID that an administrator must sift through. Even worse, permuting two `interface` blocks that span 5 lines a piece doesn't affect router behavior, but may show up as 10 deletes and 10 inserts in traditional *diff*. Figure 1 shows an example we encountered while diffing multiple versions of a configuration file for a router at Dartmouth College.

Our *bdiff* (block-based diff) can avoid this noise by directly comparing the parse trees for two versions of the same notional syntactic block. Chawathe describes a delta between two versions of hierarchical data via a *minimum cost edit script* that is defined using node insert, node delete, node update, and subtree move. For our purposes, we will assume that our syntactic blocks have unique identifiers so that we then just need to compute an edit script. System administrators could provide a flag depending upon whether *bdiff* should report permuted siblings as changes.

3 Implementation Roadmap and Proposed Evaluation

This section provides a roadmap for the implementation and proposed evaluation of our next-generation grep and diff tools that we call *bgrep* and *bdiff*. In the full version of the paper, we will work with practitioners to implement the full suite of tools we propose. For this extended abstract, we discuss our tools in the specific context of network configuration management.

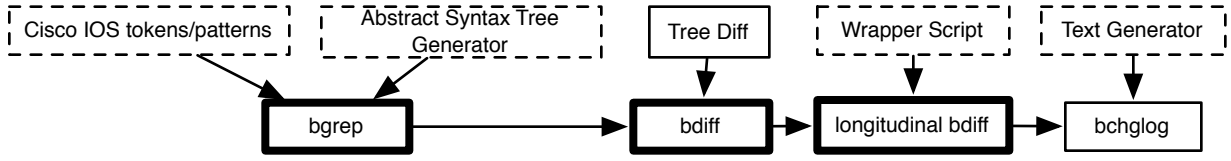

Figure 2: Our implementation roadmap, show building blocks completed and in progress, as well as the full toolsuite we plan.

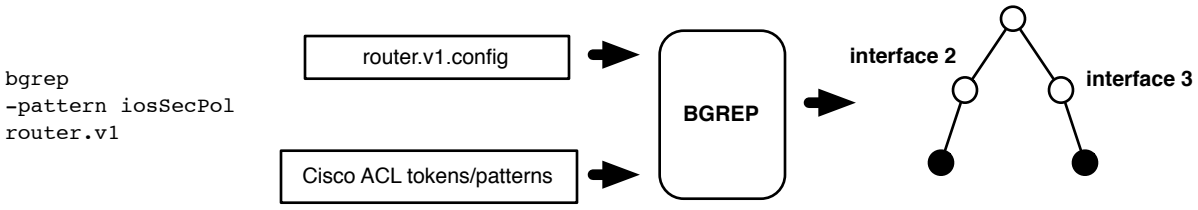

Figure 3: Using *bgrep*, a network administrator may extract blocks of Cisco IOS related to "security policy" within a given router (router.v1). Here, the administrator finds instances of the `service-policy` command within two of the router's interfaces. Our tool relies upon libraries of lightweight grammars defined for configuration languages, and outputs configuration blocks as parse trees.

Figure 2 shows our current implementation roadmap for *bgrep* and *bdiff*. First consider our roadmap for *bgrep* which consists of a library of tokens and patterns and an abstract syntax tree generator. Already we have worked with practitioners to identify patterns related to security policy in network configuration management (Cisco IOS) but plan to finish encoding these patterns in a library by the time of the full paper. We would also like to work with practitioners to write patterns for another, but very different language used by system administrators. Although we have prototype abstract syntax tree generator for entire Cisco IOS files, we plan to generalize and polish our approach for the final paper.

Our roadmap for *bdiff* is even more direct. We already have implemented a tree difference engine, but we plan to experiment with various tree edit distance algorithms before the final paper. Once we have the ability to compare *two* versions of a file, we can write a small script that queries a versioned repository of files to implement longitudinal diff mode within *bdiff*.

We see these tools as the first step towards a broader suite of block-oriented UNIX tools. For example, *bchglog* could programatically generate human-readable edit scripts generated by our *bdiff* tool. This tool, while beyond the scope of our current work, would remedy the prevalence of meaningless comments when committing changes to revision control systems that include: "asdf", "test", and "Initial revision" [16].

Furthermore, we may be able to eventually extend *bdiff* to compare two notionally similar constructs represented in two different languages. For example, a network administrator that is moving a subnet from the border routers (that use Cisco IOS) to behind the enterprise firewall (on routers that use ASA) could diff the old and new configuration files to ensure that she migrated all of the ACLs.

Bgrep Our *bgrep* is a general tool that will enable practitioners to extract blocks of code or text from a file that are nested arbitrarily deep. For example, network administrators may be interested in extracting the constructs related to "security policy" within any network interface on a given router.

Figure 3 illustrates how the system administrator can use *bgrep* to determine which interfaces implement constructs related to security policy. First, the administrator defines a library of patterns related to her notion of "security policy" in Cisco IOS; this library includes how to tokenize (via Lex) and parse (via Yacc) constructs that may include `service-policy`, and `policy-map`. The reference to the pattern library and router file is passed to *bgrep* which parses the configuration file and emits a parse tree whose leaves are blocks of code that match security constructs. In the figure above, network interfaces 2 and 3 both contain the `service-policy` command.

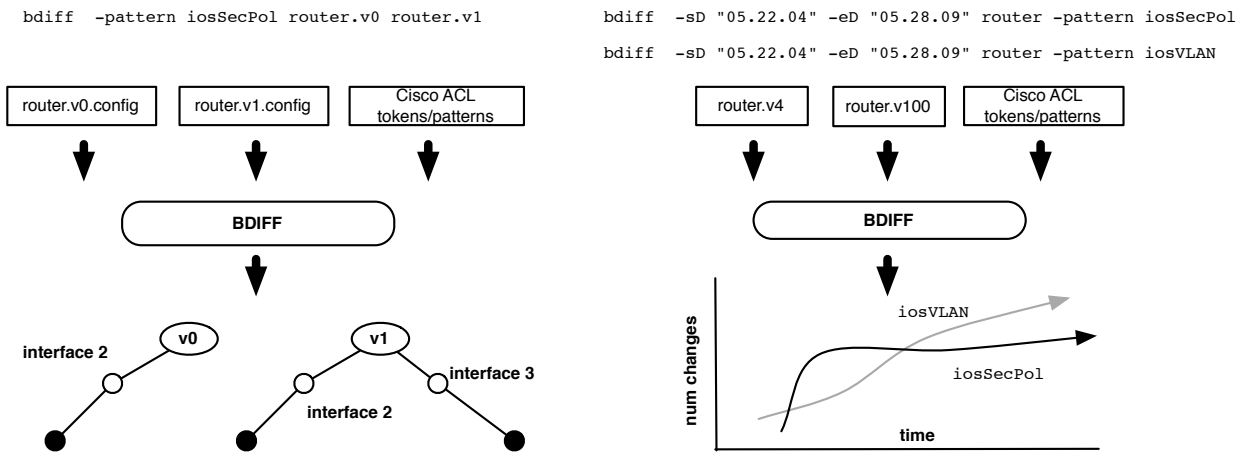

Figure 4: Using *bdiff*, a network administrator may track differences in the implementation of security policy across multiple versions of the same router. Here, the administrator may either specify two versions of the same router to compare or sequentially compare versions of the router across time via longitudinal diff mode.

Evaluation In the full version of this paper, we will work with several real-world administrators to evaluate the utility of *bgrep*. For network administration, we have already been working with Dartmouth College's Computing Services to find out their needs first-hand. In addition, they have given us access to approximately 5 years of network configuration data for the campus network. This will allow us to query the network at arbitrary points in time to count how many interfaces were configured with security policy constructs.

In addition, we have also contacted a system administrator at Tufts who uses Puppet to manage his infrastructure (a former coworker of the first author). We anticipate this to be a rich source of feedback on how *bgrep* helps administrators understand and track patterns within their configuration management systems.

Bdiff Our *bdiff* is a general tool that will enable system and network administrators to compare multiple versions of the same notional block of code. Network administrators, may be interested in comparing constructs related to security policy within two versions of a file to see how they changed. Additionally, administrators may want to measure how frequently security policy changed over a span of time using our *longitudinal diff* feature.

Figure 4 illustrates how a network administrator can use *bdiff* to directly measure how a router's implementation of a security policy changed between two versions of that router's configuration file. As before, the administrator defines (or downoads) a library of patterns relate to security policy in Cisco IOS and also passes in a sequence of files to compare. The, command returns an edit script that consists solely insertions, deletions, and moves of configuration blocks that matched security policy patterns.

Our figure also illustrates how a network administrator may use *bdiff* to generate reports of how Cisco IOS blocks relating to security policy and VLANs changed over time. Using a version-control system like subversion, we can retrieve a sequence of configuration files for a given router ordered in time. We then can do pairwise comparisons of the parse trees generated, count the number of edits per comparison, and graph the trends in time. Since these functions are piecewise linear, we can even use the area under these curves to calculate the relative amount of "work" needed to maintain VLANs versus "security policy".

Evaluation As mentioned above, the full version of this paper will include several real-world network and system administrators to evaluate the utility of *bgrep*. Already though, our collaboration with Dartmouth College's Computing Services has helped us in developing this next-generation diffing tool.

We see *bdiff* as a potential answer to the problem of understanding and maintaining a network security policy. Security policies link high-level security goals to system behavior and must evolve in order for a

system to remain secure [24]. As router configuration files evolve, however, their referential complexity increases, and with it, their chance of misconfiguration [2].

Already, we have designed *bgrep* to solve a problem faced by real practitioners. Although network administrators implement security goals within router ACL policies and firewalls, to the best of our knowledge, no tools enable administrators to easily compute how much time they spend on "policy" or how the implementation of policy has changed across time.

4 Related Work

To the best of our knowledge, we are the first to reconsider traditional UNIX tools given the increase in higher-level languages that include non-regular constructs (such as blocks nested at arbitrary depth). Certainly others have seen the need to programmatically extract and compare blocks of text. We consider XSLT, and XML-based diffing tools within this realm of research [8, 20]. However, the variety of languages seen by system administrators requires a more general approach; for example Ruby is not an XML format but it is useful to express resources in Bcfg2.

Hierarchical and Longitudinal Change As mentioned before, the *bdiff* tool we propose leverages work done by Chawathe [6].

Our work also applies Zhao et al.'s 2005 best student paper "XML Structural Delta Mining," which argues the benefits of mining change patterns in XML documents [25]. Our *bdiff*'s "longitudinal diff" mode generalizes this approach to parse trees. Additionally, Bottcher et al. introduce the new paradigm of *Change Mining* as "data mining over a volatile, evolving world with the objective of understanding change" [4].

More specifically, longitudinal studies of network configuration files has been a topic of interest since around 2009. Benson et al. demonstrated that the referential complexity of networks increases over time [2]. Sung et al. mined associations between blocks of Cisco IOS configurations including [19]. Plonka et al. explored the viability of software engineering techniques within the domain of network configuration [16]. Others that have done longitudinal studies of router configurations include Sun et al. [18] and Chen [7].

Tools for Security Policy Change Management This work fits into our broader pattern of prior work in building tools to enable human users to better manage policies of various types. Security policies range from human-readable normative reference works to lower-level router configuration files. We have developed tools that work with both.

Our previous work with natural-language policies focused on PKI certificate policy formalization [21, 23]. Our *hierarchical* model of policy is based on over 20 years of experience to model, reference, and retrieve Classical texts [9, 11, 22]. Others [3, 17, 5, 12] have experimented with formalizing high-level policy, but at the expense of necessary ambiguity (for legal purposes) and human-readability.

Our previous work with lower-level security policies expressed within router configuration files appeared this year in USENIX HotICE [24]. As mentioned previously, others have done longitudinal studies of network configuration, but have not focused on the application to managing changes within security policy. Furthermore, to the best of our knowledge, we are the first to observe that practitioners express security policy in multiple layers that must stay synchronized and must change to maintain security. Our ongoing work, therefore, focuses on tools that can detect change across multiple languages that a network or system administrator may encounter.

5 Conclusion

We have just introduced *bdiff* and *bgrep*, tools that operate on *syntactic blocks* of code and text instead of the line (the traditional information container in UNIX). Despite the variety of programming languages that system and network administrators may encounter, *syntactic blocks* remain a general interface through which one may extract and compare high-level information from multi-versioned files. Furthermore, "longitudinal diff" presents a powerful paradigm for understanding the evolution of individual components

of a network or other system. We believe the design of our tools to be truly general purpose and equally applicable to other block-based languages such as those defined by normative reference documents or even interfaces within source code.

References

[1] A. Tsalolikhin. Configuration Management Summit. *USENIX login;*, 35:104–105, 2010.

[2] T. Benson, A. Akella, and D. Maltz. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 335–348. USENIX Association, 2009.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[4] M. Bottcher, F. Hoppner, and M. Spiliopoulou. On Exploiting the Power of Time in Data Mining. *ACM SIGKDD Explorations Newsletter*, 10(2):3–11, 2008.

[5] V. Casola, A. Mazzeo, N. Mazzocca, and M. Rak. An Innovative Policy-Based Cross Certification Methodology for Public Key Infrastructures. In *EuroPKI*, 2005.

[6] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504. ACM, 1996.

[7] X. Chen, Z.M. Mao, and J. Van der Merwe. Towards Automated Network Management: Network Operations Using Dynamic Views. In *Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management*, pages 242–247. ACM, 2007.

[8] Cobena, G. and Abiteboul, S. and Marian, A. XyDiff Tools detecting changes in XML documents, 2002.

[9] Gregory Crane. The Perseus Digital Library. Retrieved May 29, 2009 from `http://www.perseus.tufts.edu/hopper/`.

[10] D. McIlroy, June 2011. Conversations about UNIX.

[11] C. Dué, M. Ebbott, C. Blackwell, and D. Smith. The Homer Multitext Project, 2007. Retrieved May 29, 2009 from `http://chs.harvard.edu/chs/homer_multitext`.

[12] J. Jensen. Presentation for the CAOPS-IGTF session at OGF25, March 2009.

[13] SVG, Markup Languages, and the Chomsky Hierarchy. Retrieved June 15, 2011 from `http://redsymbol.net/articles/svg-markup-chomsky/`.

[14] R.C. Miller. *Lightweight Structure in Text*. PhD thesis, Citeseer, 2002.

[15] Patterson, M. and Sassaman, L. Exploiting the Forest with Trees. *Blackhat*, 2010.

[16] D. Plonka and A.J. Tack. An Analysis of Network Configuration Artifacts. In *Proceedings of the 23rd Conference on Large Installation System Administration (LISA)*, page 6. USENIX Association, 2009.

[17] R. Grimm and T. Hetschold. Security Policies in OSI-Management Experiences from the DeTeBerkom Project BMSec. *Computer Networks and ISDN Systems*, 28:499, 1996.

[18] X. Sun, Y.W. Sung, S.D. Krothapalli, and S.G. Rao. A Systematic Approach for Evolving VLAN Designs. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[19] Y.W. Sung, S. Rao, S. Sen, and S. Leggett. Extracting Network-Wide Correlated Changes from Longitudinal Configuration Data. *Passive and Active Network Measurement*, pages 111–121, 2009.

[20] Y. Wang, D.J. DeWitt, and J.Y. Cai. X-diff: An effective change detection algorithm for xml documents. 2003.

[21] G. Weaver, S. Rea, and S. Smith. A Computational Framework for Certificate Policy Operations. In *Public Key Infrastructure: EuroPKI 2009*. Springer-Verlag LNCS., 2009.

[22] G. Weaver and D. Smith. Canonical Text Services (CTS). Retrieved May 29, 2009 from `http://cts3.sourceforge.net/`.

[23] G.A. Weaver, S. Rea, and S.W. Smith. Computational Techniques for Increasing PKI Policy Comprehension by Human Analysts. In *Proceedings of the 9th Symposium on Identity and Trust on the Internet*, pages 51–62. ACM, 2010.

[24] Weaver, G.A. and Foti, N. and Bratus, S. and 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*. USENIX Association, 2011.

[25] Q. Zhao, L. Chen, S.S. Bhowmick, and S. Madria. XML Structural Delta Mining: Issues and challenges. *Data & Knowledge Engineering*, 59:627–651, 2006.