XUTools: Unix Commands for Processing Next-Generation Structured Text

Gabriel A. Weaver
Dartmouth College

Sean W. Smith
Dartmouth College

Keywords: Text processing, Configuration Management, Change Management, Automation, Tools, Unix

Abstract

Traditional Unix tools operate on sequences of characters, bytes, fields, lines, and files. However, modern practitioners often want to manipulate files in terms of a variety of language-specific constructs—C functions, Cisco IOS interface blocks, and XML elements, to name a few. These language-specific structures quite often lie beyond the regular languages upon which Unix text-processing tools can practically compute. In this paper, we propose eXtended Unix text-processing tools (xutools) and present implementations that enable practitioners to extract (`xugrep`), count (`xuwc`), and compare (`xudiff`) texts in terms of language-specific structures. We motivate, design, and evaluate our tools around real-world use cases from network and system administrators, security consultants, and software engineers from a variety of domains including the power grid, healthcare, and education.

1 Introduction

During our fieldwork, we observed the need to generalize Unix text processing tools so that practitioners can process the right type of string for the job at hand. We thus need to extend traditional Unix tools because many modern, structured-text formats break assumptions of traditional Unix tools.

Traditional Unix tools operate on sequences of characters, bytes, fields, lines, and files. When lines and files correspond to language-specific constructs, traditional Unix tools work well. For example, lines of Apache log files correspond to HTTP requests.

However, there are many other language-specific constructs besides the line. Many file formats found in markup, configuration, and programming languages encode meaningful structures via nested blocks. Natural-language documents may be divided up into chapters, paragraphs, and sentences. Sentences themselves may be further parsed. Configuration files and programming languages, similarly, may be divided up into blocks of code. For example, Cisco IOS interface blocks or C function blocks are information units that network administrators and developers reference daily.

The prevalence of multi-line, nested-block-structured formats has left a capability gap for traditional tools. Today, if practitioners want to extract interfaces from a Cisco IOS router configuration file, they must craft an invocation for `sed(1)`

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

Using our xutools, practitioners need only type

```
xugrep '//ios:interface'
```

We designed and built eXtended Unix text-processing tools (xutools) *so that practitioners could process files in terms of the language constructs appropriate to the problem at hand—even if these languages lie beyond regular expressions.*

Besides "eXtending Unix," we also chose the *xu* prefix from the Greek word ξυλον, denoting "tree" or "staff." We find the first sense especially appropriate given that xutools operate on parse trees and process texts (traditionally printed on trees). The second sense is appropriate because xutools are designed to support IT staff in their real-world needs. Finally, ξυλον comes from the word ξυω , meaning to scrape, and our xutools, particularly `xugrep`, are well-suited to scraping content from document trees.

In order to support a variety of languages, our xutools rely upon a modular grammar library, as well as *xupath*, a general purpose querying language for structured text, which we based upon XPath. Although xutools may eventually encompass a broad range of Unix tools, this paper

presents xutools that extract (`xugrep`), count (`xuwc`), and compare (`xudiff`) structured text formats.

Our xutools generalize the class of languages that we can practically process on the command-line. Recall (e.g., [33]) that a **language** is a set of strings and a **regular language** is a set of strings that can be recognized by some finite automaton or equivalently, by a regular expression. In other words, we can write a regular expression to recognize whether a string is in a given language if and only if that language is regular. In contrast, a **context-free** language is a set of strings that can be recognized by a finite automaton with a stack. In practice, this means that we can write a context-free grammar to recognize a string in a context-free language. All regular languages are context-free (they just don't use the stack)—but not vice-versa. For example, the language of all strings with properly-nested parentheses is context-free but not regular. One cannot solve the **parenthesis-matching problem** with regular expressions.

In traditional Unix text-processing tools, most of the types of strings (bytes, fields, characters, files, lines) are either directly matched or indirectly split via a regular expression. In contrast, many of the languages in which practitioners are interested (XML elements, JSON subtrees, Cisco IOS interfaces, C function blocks), are not regular—but are context-free. This is no coincidence since the syntax of many programming languages was traditionally specified via context-free grammars. Context-free languages allow practitioners to recognize strings that possess a recursive or hierarchical structure. Furthermore, as languages evolve and acquire new constructs, a grammatical description of a language is easily extended (even for regular languages) [1].

Furthermore, Unix text-processing tools tend to operate on all the lines in a given input stream. For example `grep(1)` reports lines that contain a substring that matches a regular expression, `diff(1)` reports all differences between two files, and `wc(1)` counts all of the bytes, characters, words or lines in a file [32, 38, 39]. Often, however, practitioners are interested in reporting a matching line in terms of the block in which that match occurs; they are interested in reporting line-level differences between two router configurations in terms of interface blocks; or they are interested in counting the lines within each function block in C source. We extend Unix tools to report results in terms of contexts other than the file.

This Paper: We motivate (Section 2), design and implement (Section 3), evaluate (Section 4), and show the novelty of (Section 5) our tools relative to real-world examples. Table 1 illustrates our path through each of these sections. Finally, in Section 6 we discuss future work and in Section 7 we conclude.

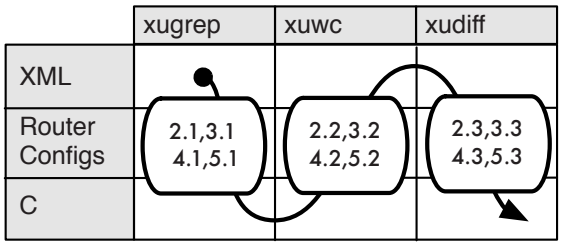

Table 1: For each section of our paper, we will discuss each of our tools relative to one or more real-world use cases that use the following structured-file formats.

2 Motivating use cases

During our fieldwork we observed the need to generalize Unix text-processing tools. Specifically, we worked with various network and system administrators and auditors of major electrical power utilities in the United States and discussed their challenges. We have also met with network administrators at Dartmouth College. In addition, we received much feedback from a poster on our tools [42] and recorded and analyzed these interactions at `http://www.xutools.com/`. As a result, we have discussed our work with practitioners from the RedHat Security Response team. Finally, we have met with the CEO of a network security company based in Germany regarding uses for our tools.

The intent of xutools is to help practitioners process the right type of string for the job at hand. Users (such as network and system administrators, security consultants, and software engineers) from a variety of domains (including the power grid, healthcare, and education) have emphasized the need for tools that let them operate on strings in languages that are relevant to them.

Many interesting strings are not in regular languages. Current Unix text-processing tools do not allow practitioners to operate on blocks of text nested arbitrarily deep or to report results with respect to that nesting. However, many practitioners must process these blocks in order to solve problems they encounter.

Figures 1 and 2 illustrate the mismatch between strings of interest in modern file formats and the strings upon which current Unix text-processing tools operate.

2.1 Use cases for rethinking `grep(1)`

NVD-XML: Very recently, practitioners at the RedHat Security Response Team wanted a way to grep the XML feed for the *National Vulnerability Database (NVD)*.[1] Specifically, they wanted to know how many

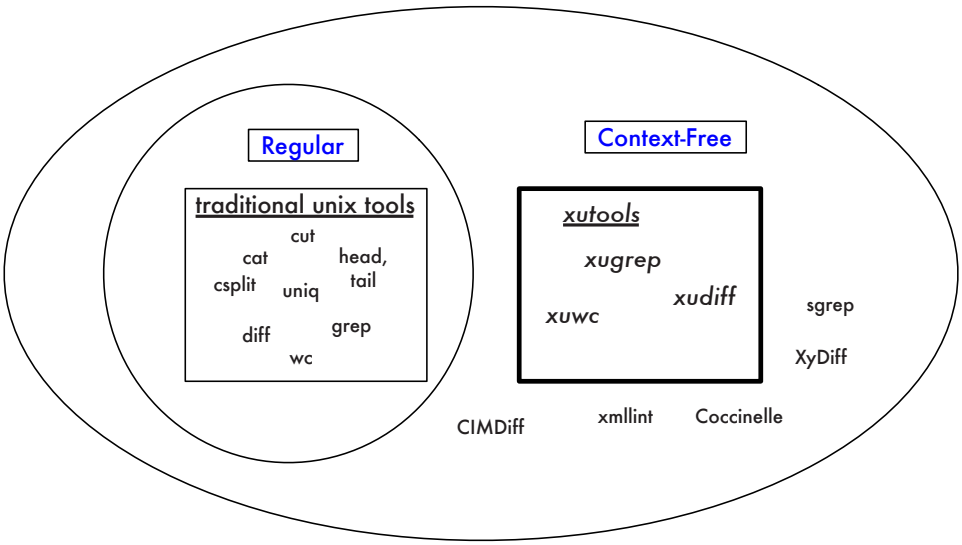

Figure 2: Our xutools improve on prior work by systematically extending Unix tools to operate on the broader class of languages that are encountered in modern file formats.

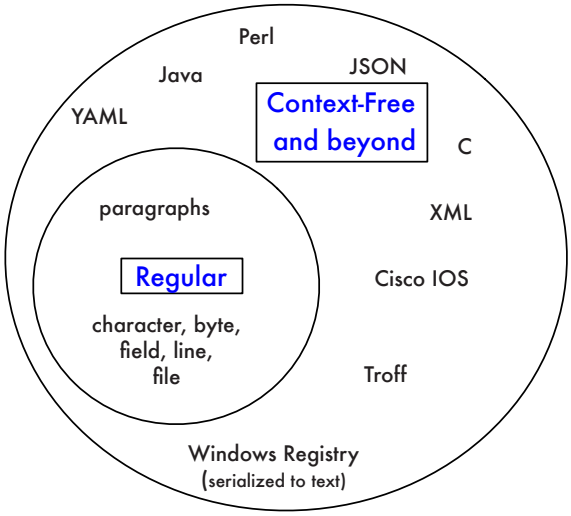

Figure 1: The languages upon we which we need to operate are now more general than can be described with regular expressions.

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 parenthesis-matching problem. This limitation motivates the capability to be able to report matches with respect to a given context.

In contrast, we need a `grep(1)` that can handle strings in context-free languages because when we extract XML elements, we want to ensure that every opening tag is matched by a closing tag. Multiple XML elements may share the same closing tag, and XML elements may be nested arbitrarily deep. Therefore, we need parenthesis matching to recognize XML elements. We need a `grep(1)` that can report matches with respect to the contexts defined within the NVD-XML vocabulary. (Although `xmllint(1)`'s shell-mode grep certainly provides one solution, it is not general enough to deal with languages other than XML [43]. We will discuss `xmllint(1)` in more detail in Section 3.)

C: 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" [30, 42][2]. Traditional `grep(1)` cannot do this.

Alternatively, there may also be other constructs not explicitly defined via a standard grammar for a file format or language (such as ANSI C). For example, if we

could specify a context-free grammar for a C patch, then we could extract sections of code that are similar to that patch. Practitioners at the RedHat Security Response Team, for example, could use this to find all locations of unpatched code within embedded libraries scattered throughout source distributions.

Traditional `grep(1)` cannot handle these use cases. These use cases require us to solve the parenthesis-matching problem and also motivate a tool that can report matches with respect to a given context.

A regular expression that directly matches a C block cannot be created because the language of C blocks is not regular. Specifically, brackets close C functions, but they also close other kinds of blocks (such as if-statements) and so without matching, the closing bracket is ambiguous.

Secondly, these examples illustrate the benefits of a tool that can report matches with respect to the contexts defined within the C grammar. Practitioners need a suitable tool to report lines that contain a call to `malloc`, or even functions that contain the match.

2.2 Use cases for rethinking `wc(1)`

Router Configuration Files Network administrators want tools to understand their network configuration because many errors are network configuration errors [23, 35]. The network configuration literature attests that administrators may want to see how those network configurations change across time [26, 34], but no tools are available for admins to quickly and easily perform longitudinal studies on their own network data.

The usual metrics employed during longitudinal studies of network configuration data, such as lines of configuration, are general purpose but do not take other, language-specific measures into account. One exception here is Plonka et al. who look at stanzas. A stanza is a "set of adjacent related lines, or a paragraphs of configurations with a common purpose," such as an interface definition [26].

Network administrators configure and maintain language-specific constructs, such as interfaces, and as such, they would like to be able to measure their configuration files relative to these language-specific 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 interfaces within the set of campus routers use a particular active VLAN.

Traditional `wc(1)` cannot handle this use case. Traditional `wc(1)` counts lines in a file and so can calculate lines of configuration. Language-specific measures of configuration files, however, need a tool that can parse and count relative to language-specific constructs. `wc(1)` only counts languages tied to physical units of storage such as bytes, characters, words, and lines.

We need a `wc(1)` that can solve the parenthesis-matching problem. Practitioners want to count how many lines there are per interface block within a router configuration file. Since in general the set of block-based constructs in Cisco IOS is a context-free language, we must make the Unix `wc(1)` utility aware of context-free languages. (In fact, we could accomplish this by using our `xugrep` above to extract the interfaces in document order, escape the newlines in each block, and pipe the sequence of interfaces into `wc -l`.)

2.3 Use cases for rethinking `diff(1)`

Router Configuration Files Current tools such as *Really Awesome New Cisco config Differ (RANCID)* [28] 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.

For example, if a network administrator moves a network interface for a router configuration file, then an edit script for the router configurations in terms of lines may report the change as 8 inserts and 8 deletes. However, an edit script that compares configurations in terms of interfaces ("interface X moved") might be more readable and less computationally intensive.

Traditional `diff(1)` cannot handle this use case because it requires us both to solve the parenthesis-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 [6]. 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(1)` (where each line corresponds to an interface or line in an interface).

However, we want to understand how the lines in a configuration change with respect to the contexts defined by the Cisco IOS language. In this use case, we want to report changed lines in the context of their containing interface blocks. Alternatively, we could also choose to

report changes to interface blocks themselves.

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 (see above), we offer a general-purpose solution for a wider variety of structured texts.

3 Design and implementation of our tools

We now present the design and implementation of each of our xutools: `xugrep`, `xuwc`, and `xudiff`. Our tools are intended to provide extended versions of `grep(1)`, `wc(1)`, and `diff(1)` respectively [32, 38, 39]. For each tool, we will give examples of the command syntax, describe the algorithm upon which the tool is based, and discuss our current implementation.

Before we discuss the tools, however, we will first present the design and implementation of two common components shared among all of our tools: a library of language grammars, and xupath, an XPath-like querying language for structured text in general (not just XML) [45].

3.1 Grammar library

The goal of our eXtended Unix text-processing tools (xutools) is to allow practitioners to operate on strings in languages that are relevant to them. Therefore, we want each of our grammars to be small and lightweight.

Design Although system administrators 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. Just as C developers need to know the name, purpose, and calling sequence of a function, but not its implementation, in order to use that function in their own code, so do our users need to know the name of a production in our grammar library, not how the production is written, in order to process the corresponding construct with xutools. Since our grammar library's production names are aligned with the constructs practitioners use in practice, we expect our interface to the grammar library will be relatively easy to learn. For example, the production name for an interface in the Cisco IOS grammar is IOS:INTERFACE while the production name for a section in a document encoded using the *Text Encoding Initiative's XML guidelines (TEI-XML)* [5] is TEI:SECTION.

Implementation We implement our grammars using the PyParsing library [21]. We have currently implemented small grammars for subsets of two XML vocabularies (NVD-XML and TEI-XML), and for Cisco IOS. We have implemented an xupath grammar based upon Zazueta's Micro XPath grammar [47]. In addition, we are also using McGuire's subset-C parser [20]. Finally, we have a BUILTIN grammar for commonly-used, general-purpose constructs such as lines.

3.2 xupath

Our goal for xupath is 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 set of high-level-language constructs that they want to process. Elements in an xupath query result set have the following fields: *text*, *production-name*, and *label*. The value of an element's *text* field is the string extracted from the text. The value of an element's *production-name* field is the sequence of productions applied to extract that string. Finally, the value of an element's *label* field depends upon the grammar. In our TEI-XML security policies, the value of a *label* is that node's passage header, for example *Introduction*. In our Cisco IOS security policies, the label corresponds to the network primitive's name such as *interface GigabitEthernet0/2*.

Design Our eXtended Unix text-processing tools generalize traditional Unix tools to (1) match context-free strings in the language of a context-free grammar, and (2) describe the context in which processing and reporting should occur. We observed that the XPath Query language [45] performs this function for XML documents. Therefore, we use an XPath-like syntax to express our queries on texts. This design decision offers us the additional benefit of not having to re-invent the wheel when thinking about how to select nodes from a set of parse trees.

Consider, for example, our first use case from the RedHat Security Response team. The team member wanted to know (1) how many NVD entries contain the `cpe:/a:redhat` string, (2) the vulnerability score of these entries, and (3) how many elements in the NVD-XML feed contain the `cpe:/a:redhat` string.

If all of our queries were on XML document trees, then we could answer these by using XPath within `xmllint`. For example, we could answer the third question by issuing the `grep a:redhat` command within the `xmllint` shell.

Practitioners, however, want to process Cisco IOS, C, JSON, or other languages (see Figure 1) and none of these languages are XML. Our xupath provides XPath-like functionality to context-free languages.

Consider these examples. Just as the XPath //DEFAULT:ENTRY selects all entry elements in the DEFAULT namespace, so does our xupath //NVD:ENTRY select all strings in the language of the ENTRY production within the NVD grammar. If we are concerned about namespaces, then we can prefix the production name with the appropriate namespace within the NVD grammar. Just as the XPath

```
//DEFAULT:ENTRY /CPSS:SCORE
```

references all score elements from all entry elements, so does our xupath

```
//NVD:ENTRY/NVD:SCORE
```

select all strings in the language of the score production from the set of all strings in the language of the entity production in the NVD grammar. Just as the XPath

```
//DEFAULT:ENTRY
[.//*[contains(*,'cpe:/a:redhat')]]
```

selects all entry elements whose descendants contain the string `cpe:/a:redhat`, so does our xupath

```
//NVD:ENTRY
[re:testsubtree('cpe:/a:redhat')]
/NVD:SCORE
```

select all strings in the language of the entry production in the NVD grammar that contain a match to the regular expression `cpe:/a:redhat`.

Our xupath generalizes such queries beyond XML to context-free languages in general such as subsets of Cisco IOS and C.

In Cisco IOS, we can use an xupath to reference all strings in the language of the interface production in our Cisco IOS grammar: //IOS:INTERFACE. We can also use an xupath to select all interfaces that contain an access-group:

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

The xupath syntax also gives us an easy way to answer queries such as *where is acl 23 used?*.

In C, we can use an xupath to reference all strings in the language of the function production in a C subset grammar: //CSPEC:FUNCTION. We can also, however, grab all the lines within those functions with an xupath that mixes productions from the C specification grammar and our BUILTIN grammar: //CSPEC:FUNCTION/BUILTIN:LINE.

Implementation: Since the xupath syntax is based upon XPath, we implemented xupath in Python as a modified MicroXPath [47] grammar ported to PyParsing [21]. Given an xupath query, our grammar generates a parse tree of six types of nodes. Xupath parse trees are implemented using Python dictionaries. Consider the following examples in NVD and C shown in Figure 3.

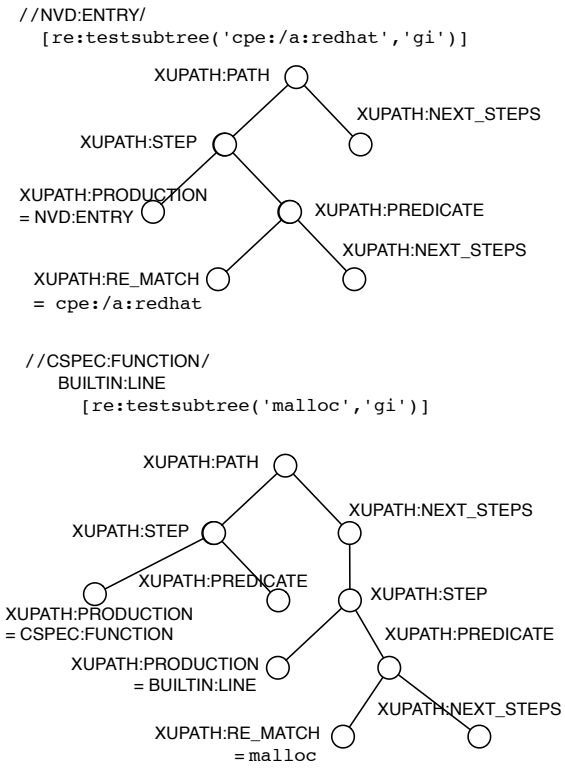

Figure 3: Two sample parse trees for xupath queries on NVD-XML and C.

3.3 xugrep

Our `xugrep` generalizes `grep(1)`; `xugrep` extracts all strings in a set of files that match an xupath.

Design: Traditional Unix `grep(1)` extracts all lines in a file that contain strings in the language of a regular expression. `Grep(1)` outputs each line containing a substring that matches a regular expression. The positions of those strings are line numbers.

Our `xugrep` generalizes the class of languages that we can practically extract on the Unix command-line from regular to context-free. A call to `xugrep` reports all strings that satisfy the given xupath within the context of the input files. The `-l` option tells our tool to

xugrep example input/output

```
bash-ex1$ xugrep -1 "//nvd:entry[re:testsubtree('cpe:/a:redhat','gi')]/nvd:score"
        nvdcve-2.0-2012.xml

file_path                 nvd:entry       nvd:score       region
nvdcve-2.0-2012.xml       CVE-2012-2110   1               <cvss:score>7.5</cvss:score>

bash-ex2$ xugrep -1 "//cspec:function" example.c

file_path  cspec:function  region
example.c  putstr          int/nputstr(char *s)/n{/n    ... putchar(*s++);/n}
example.c  fac             int/nfac(int n)/n{/n    if (n == 0)  ... n*fac(n-1);/n}
example.c  putn            int/nputn(int n)/n{/n if (9 < n)... putchar((n%10) + '0');/n}
example.c  facpr           int/nfacpr(int n)/n{/n   ...  putstr("\n");/n}
example.c  main            int/nmain()/n{/n    int i;/n ... return 0;/n}
```

Table 3: We can use `xugrep` on XML files as well as C source files to extract language-specific regions of text.

xugrep usage

```
xugrep [-1] <xupath> <input_file>+
```

Table 2: Our `xugrep` will report all strings that satisfy the given xupath within the context of the input files. Results may be reported in a one-line-per-match format via the `-1` option.

escape newlines within each of the result strings so that each match fits neatly on a single line.

The use cases of Section 2 motivate the need to be able to extract strings ranging from XML elements to C function blocks. We designed xugrep for practitioners to satisfy both these requirements as shown in Table 3.

Algorithm: `xugrep` generates a result set for the xupath query as it traverses the query's parse tree. As we traverse the xupath parse tree, we create a set of elements whose corresponding strings (stored in each element's *text* field) satisfy the query. We update the query result set when we encounter nodes whose production names are XUPATH:PRODUCTION and XUPATH:RE_MATCH in the xupath parse tree. When we encounter an XUPath parse-tree node whose *production-name* has the value XUPATH:PRODUCTION, we update the query result set to only include strings in the language specified by the production specified by the grammar corresponding to that parse-tree node. Similarly, when we encounter an XUPATH:RE_MATCH node, we update the query result set to only include strings that contain matches in the language of the corresponding regular expression for that node.

Implementation: We implemented `xugrep` in Python using a functional programming style. We have one function for every type of xupath parse tree node. Our current implementation of `xugrep` is 191 lines.

3.4 xuwc

Our `xuwc` generalizes `wc(1)` to count the number or size of strings in an xupath result set relative to some context.

Design: As stated by the Unix man pages, traditional `wc(1)` counts the number of words, lines, characters, or bytes contained in each input file or standard input [32]. Our `xuwc` generalizes `wc(1)` to count strings in context-free languages and to report those counts relative to language-specific contexts.

First, `xuwc` lets practitioners match strings in context-free languages. Section 2 reported how practitioners may want to count the size and number of language-specific structures within a Cisco IOS router configuration, C source file, or some other language. In the first two examples in Table 5 (**bash-ex1** and **bash-ex2**), `xuwc` allows practitioners to perform these tasks. `xuwc` does not have `wc(1)`'s `-clmw` flags because practitioners can specify their own structures to count (including bytes, lines, characters, and words) via the `--count` option.

Second, `xuwc` generalizes `wc(1)` to report counts relative to language-specific contexts. The Unix `wc(1)` utility counts the number of bytes, characters, words, or lines *relative to a file*. However, instead of reporting the number of lines in each router configuration file, we might prefer to report the number of lines in each interface. Alternatively, we may also want to count the number of entries in an *Access Control List (ACL)*. The last two examples in Table 5 (**bash-ex3** and **bash-ex4**) illustrate

xuwc usage

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

Table 4: Given an xupath and a set of files, `xuwc` will count all matches in the result set in the context of the file.

xuwc example input/output

```
bash-ex1$ xuwc "//cspec:function" example.c
5       cspec:function  example.c
5       cspec:function  1       file_path       TOTAL

bash-ex2$ xuwc "//ios:interface/builtin:line" router.example
37      builtin:line    router.example
37      builtin:line    1       file_path       TOTAL

bash-ex3$ xuwc --context=ios:interface "//ios:interface/builtin:line" router.example
8       builtin:line    router.example.GigabitEthernet4/1
3       builtin:line    router.example.Null0
18      builtin:line    router.example.GigabitEthernet4/2
8       builtin:line    router.example.Loopback0
37      builtin:line    4       ios:interface   TOTAL

bash-ex4$ xuwc --count=builtin:byte --context=builtin:line "//ios:interface/builtin:line"
         router.example
        ...
24      builtin:byte    router.example.GigabitEthernet4/2.10
31      builtin:byte    router.example.GigabitEthernet4/2.11
13      builtin:byte    router.example.GigabitEthernet4/2.12
761     builtin:byte    37      builtin:line    TOTAL
```

Table 5: Four examples of our `xuwc` tool applied to C source code and Cisco IOS configuration files. Note our ability to *drill down* into the sizes of interfaces and then lines in Examples 3 and 4 (**bash-ex3** and **bash-ex4**) respectively.

the former scenario as well as the ability to combine these two generalizations of counting and context.

Algorithm: We implement `xuwc` by processing an `xugrep` report for the provided xupath. By default, if the `--count` parameter is unspecified, `xuwc` counts the number of strings that are in the language of the last production name in the xupath. If `--context` is unspecified, then `xuwc` reports counts relative to the entire file.

Our `xuwc` proceeds as follows. First, we call `xugrep` on the xupath to obtain a query result set. One of two cases follows next.

In the first case, we count the number of strings in the languages of the following productions–BUILTIN:BYTE, BUILTIN:CHARACTER, BUILTIN:WORD–or a regular expression, then we iterate through each element in the query result set. For each element, we scan the string stored in the element's *text* field and count the number of matches in the language of the production. Note that the context here *must* be that of the last production name in the xupath since these are the only strings stored in the result set elements.

Otherwise, we are counting strings in the language recognized by some other grammar production. This production name must be in our xupath or else an exception is thrown. For each element in the query result set, we scan the string stored in the element's *text* field and count the number of matches in the language of the production relative to the context in which it occurs.

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

3.5 `xudiff`

`xudiff` generalizes `diff(1)` 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.

Design: Traditional Unix `diff(1)` computes an edit script between the sequences of lines in a file. `diff(1)` 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 (insert, delete, update) [38]. All of these edit operations are performed upon *lines* in the context of the *entire file*.

While traditional `diff(1)` lets practitioners compare files in terms of the line, our `xudiff` allows practitioners to be able to compare files in terms of higher-level language constructs specified by productions in a context-free grammar.

xudiff usage

```
xudiff [--count=<grammar:production>]
       <xupath> <input_file1> <input_file2>
```

Table 6: Our `xudiff` compares two files in terms of the parse trees generated by applying an xupath to each file. Practitioners may also specify units in which to calculate edit costs.

Our `xudiff` lets practitioners compare files in terms of high-level language constructs. In Section 2, we saw that practitioners need to be able to compare both sections and subsections in TEI-XML documents as well as interface blocks in Cisco IOS router configurations. We designed `xudiff` so that practitioners could easily make such comparisons as shown in Table 8. In these examples, we show the output from comparing two versions of a router configuration file in Table 7.

Our `xudiff` also allows practitioners to specify the units that they want to use to compute the cost of a change. Example 1 (**bash-ex1**) issues the same query as Example 2 (**bash-ex2**), but the first example counts the cost of changes in terms of lines (for a distance of 4) whereas the second example computes the cost of changes in terms of words (for a distance of 9). The `I`, `D`, and `U` characters stand for insert, delete, and update respectively while the first number in the square brackets corresponds to the running edit cost, and the second number to cost of a particular edit. Alternatively, if we computed the cost of changes in terms of characters, then we would see that only 3 characters were changed in the second line of the `Loopback0` interface.

Algorithm and Implementation We have implemented the Zhang and Shasha tree edit distance algorithm [48] that is the basis for our `xudiff`. As a result, we can currently compute edit scripts for parse trees in TEI-XML, Cisco IOS, and C. The above examples are based upon the edit scripts we currently compute. Our Python implementation of the algorithm is 254 lines.

4 Evaluation

In this section we evaluate each of our tools qualitatively and quantitatively. The qualitative evaluation consists of anecdotal feedback regarding our tools from real-world practitioners. The quantitative evaluation includes the worst-case time complexity of our tools, lines of code, and test coverage.[3]

In addition to the above evaluation, Section 5 compares our xutools against existing tools that handle the formats we discuss in Section 2.

router.v0.config	router.v1.config
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 !	interface Loopback0 description really cool description ip address **333.444.1.581** 255.255.255.255 no ip redirects 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** !

Table 7: The two Cisco IOS router interfaces upon which examples 2 through 4 are based. Changes are highlighted in **bold**.

xudiff example input/output

```
bash-ex1$  xudiff "//ios:config" router.v0.config router.v1.config

Distance: 4.0
I [4.0, 1.0] _.GigabitEthernet4/2.ip flow ingress (line)
D [3.0, 1.0] _.GigabitEthernet4/2.ip access-group inbound_filter out (line)
U [2.0, 1.0] _.Loopback0.crypto map azalea (line) -> _.Loopback0.crypto map daffodil (line)
U [1.0, 1.0] _.Loopback0.ip address 333.444.1.185 255.255.255.255 no ip redirects (line) -> _.Loopback0.ip
address 333.444.1.581 255.255.255.255 no ip redirects (line)
bash-ex2$  xudiff --count=builtin:word "//ios:config" router.v0.config router.v1.config

Distance: 9.0 builtin:word
I [9.0, 3.0] _.GigabitEthernet4/2.ip flow ingress (line)
D [6.0, 4.0] _.GigabitEthernet4/2.ip access-group inbound_filter out (line)
U [2.0, 1.0] _.Loopback0.crypto map azalea (line) -> _.Loopback0.crypto map daffodil (line)
U [1.0, 1.0] _.Loopback0.ip address 333.444.1.185 255.255.255.255 no ip redirects (line) -> _.Loopback0.ip
address 333.444.1.581 255.255.255.255 no ip redirects (line)
```

Table 8: Output of our `xudiff` utility when comparing the two versions of the Cisco IOS file in Table 7. Notice the ability to express the same set of changes in terms of lines (**bash-ex1**) or in terms of words (**bash-ex2**).

4.1 Grammar Library

The goal of our grammar library was to provide a common interface by which all of our xutools could parse text in terms of language-specific constructs. We recognize that system administrators want to avoid writing grammars, but desire the power and convenience of processing blocks of markup, configuration, and programming languages. Therefore, we wanted to create a modular, lightweight approach to parsing language-specific constructs.

Qualitative: 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 three strategies based upon feedback from practitioners in order to grow the library of grammars.

First we are writing a library of grammars for commonly-used languages. In last year's LISA poster session, practitioners suggested that we cover a few useful languages such as C, Java, Perl, XML, JSON, and YAML. Furthermore a repository of grammars for a variety of languages may also improve system security in the long run according to LangSec, Language-Theoretic Security, researchers [4].

Second, we could write a parser for languages used to specify grammars. 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 [19, 24, 29, 44, 46].

Finally, practitioners might be able to parse a text written in one language using a grammar for a very *similar* language. For example, if you try to open a Perl file in emacs' bash mode, then it looks decent. This *approximate matching* strategy could be a mechanism for xutools to handle languages that they haven't seen before.[4]

Quantitative: Our goal is to let practitioners write or use small, lightweight grammars to extract and process language-specific structure. The size of our grammars (shown in Table 9) illustrates that we can process texts in a variety of languages, in meaningful ways with grammars that are relatively small in terms of total number of productions, number of production names, and the number of lines needed to encode the grammar.

We currently use the PyParsing [21] library to implement our grammars and parse files according to these grammars. PyParsing implements a recursive-descent parser. In the worst-case, recursive descent parsing may require backtracking and so the time complexity may become exponential in the length of the input string (although this can be mitigated to some extent by memoization). Although we have not encountered any practical issues in using our tools on natural-language security policies and configuration files, we can improve our performance by using a different kind of parser. As stated by Aho, Sethi, and Ullman, "linear algorithms suffice to parse essentially all languages that arise in practice" [1].

Grammar	# productions	# production names	# lines
NVD-XML	6	3	8
TEI-XML	19	8	26
Cisco IOS	8	7	23
C	26	1	55
XUPath	11	0	28
Builtin	1	1	2

Table 9: The sizes of the grammars used by our xutools in terms of total number of productions, production names, and number of lines to encode the grammar.

4.2 xupath

We designed and implemented xupath so that practitioners could query a broader class of structured texts than XML. We only provide a qualitative evaluation of xupath.

Qualitative: xupath provides a consistent interface between language constructs and how those constructs are encoded within text. Encoding formats will continue to change. The SGML of the 80s is the XML of today. Although XML allows one to repurpose information, it may be an unreadable and/or bloated format for system administrators to directly edit. Furthermore, it is unnecessary (and potentially expensive in terms of resources) to convert C, Java, or some other format into XML.

Our tools use the xupath syntax to specify references to high-level language constructs so that the encoding of these constructs is transparent to the practitioner.

Since the *encodings* of meaningful constructs will change over time, we want to build tools to operate in terms of *references* to those constructs. Consider the historical transmission of text in which books and lines of Homer's Odyssey migrated from manuscripts, to books, to digital formats. The encoding of the text changed, but the constructs of book and line survived. In other words, the way in which people *referenced* the information remained stable although the encoding changed.[5]

Furthermore, formats such as CFEngine and Apache httpd configurations hint that there are structured text formats that are "in-between" traditional line-based formats and XML. In addition, there may be multiple ways to parse a text. Since our xupath syntax allows one to query a text using a sequence of production names from different grammars, practitioners can reference and parse these "in-between" formats in a variety of ways.

Finally, because xupath's syntax is based on XPath, we argue that the learning curve for xupath is lower than other querying languages might have been.

4.3 `xugrep`

`xugrep` extracts all strings in a set of files that match a xupath.

Qualitative: In Section 2 we motivated `xugrep` with two (of several) real-world use cases. We now briefly discuss how `xugrep` satisfied each of those use cases.

In Table 3 of Section 3 we presented actual output of how `xugrep` satisfies the two use cases of Section 2.

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. During our discussion of xupath, we showed that `xmllint(1)` could satisfy this use case but that our `xugrep` tool operates on a more general class of languages.

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 [30, 42].

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 XUPATH:PRODUCTION or XUPATH:RE_MATCH, we must iterate through each element in our query result set and scan for matches. As previously noted in our evaluation of our grammar library, we use the PyParsing library which is a recursive-descent parser [21]. The worst-case time complexity for recursive-descent parsing is exponential in the size of the input string [1]. Therefore, we can probably improve our practical performance by reimplementing our tools with a linear-time parser.

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

4.4 `xuwc`

Our `xuwc` allows practitioners to count the number of language-specific constructs within a file (or another language-specific context).

Qualitative: Section 2 motivates `xuwc` with a use case involving Cisco IOS. In Table 5 of Section 3 we presented actual output of how `xuwc` satisfies this use case as well as a use case involving C function blocks.

Our `xuwc` allows network administrators to perform longitudinal studies upon their router-configuration files. For example, we could look at how the total number of lines per interface evolved over multiple versions of a router configuration. Alternatively, we could chart the number of ACLs over time.

Furthermore, `xuwc` may also be useful for looking at code complexity over time. Analogous to 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.

Quantitative: We implemented `xuwc` as a routine to process the 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 in the worst case `xuwc` loops through all the elements in the result set exactly once. Therefore, the worse-case time complexity of our `xuwc`, not including the time `xugrep` takes to generate the query result set, is linear in the size of that result set.

Our Python implementation of `xuwc` is 96 lines and so we can easily port this to another language if desired. We have 408 lines of unit tests for this tool that cover examples in TEI-XML, NVD-XML, Cisco IOS, and C.

4.5 `xudiff`

Our `xudiff` allows practitioners to compare two files in terms of higher-level constructs specified by productions in a context-free grammar.

Qualitative: Section 2 motivates `xudiff` with RANCID as well as document-centric XML. In Table 8 of Section 3 we demonstrated how our `xudiff` satisfies the former of these use cases (although it currently has the capability to satisfy the latter use case as well).

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. Parse trees

encode recursive or hierarchical structure and this hierarchy often reflects levels of abstraction within a language. 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.

Quantitative: Our `xudiff` design relies upon the Zhang and Shasha algorithm to compute edit scripts. The time complexity for this algorithm is $O(|T_1| * |T_2| * min(depth(T_1), leaves(T_1)) * min(depth(T_2), leaves(T_2)))$ and its space complexity is $O(|T_1| * |T_2|)$ [48]. In these formulae, $|T_1|$ 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.

4.6 Overall xutools evaluation

Overall, we tried to design our xutools in a modular fashion so as to make them extensible and usable while avoiding needless complexity. The same grammars can be used by all of our tools. We have a simple, unified querying interface (xupath) shared among `xugrep`, `xuwc`, and `xudiff`. In the past, when we presented designs for our tools, practitioners warned us not to develop a set of command line flags that was too complex. We believe that the syntax for our current tools is straightforward.

Our xutools need not only run on Unix. Since we have implemented xutools in Python, we can also run these tools on a Windows' command prompt. In fact, we have documented and tested the full installation of an earlier version of `xugrep` on Windows. This is especially useful in the context of electrical power control systems in which Windows machines are common.

5 Related work

We now discuss our xutools in the context of prior work done in both industry and academia. We break out the relevant work discussion so as to make it clear what is novel about our work from an academic perspective as well as the gain in utility over the currently-available tools in industry. Furthermore, this section gives us an opportunity to compare our xutools against existing tools that handle the formats we discuss in Section 2.

5.1 `xugrep`

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` [18]. `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 may be used to transform and extract information based upon the structure of the XML. We have already mentioned libxml2's `xmllint(1)` [43] 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 [17].

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 [8]. 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) [27].

Pike's structural regular expressions allow users to write a program to refine matches based on successive applications of regular expressions [25]. Our approach is different because we extract matches based upon whether a string is in the language of the supplied grammar.

Academia: Although Grunschlag has built a context-free grep [16], this classroom tool only extracts matches with respect to individual lines. Coccinelle [10] 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 Section 2, 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 [34]. Similarly, Plonka et al. look at stanzas in their work [26].

5.2 xuwc

Our `xuwc` tool allows practitioners to count the number or size of a language-specific construct within an xupath result set.

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(1)`.

Industry: Certainly 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 [22]. Vil provides metrics, visualization, and queries for C#, .NET, and VisualBasic.NET and can count methods per class and lines of code [41]. 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 [11].

Finally, Windows Powershell has a nice CmdLet called Measure-Object that allows people to gather statistics about an object [27].

5.3 xudiff

The goal of our `xudiff` is to produce a general-purpose tool to compare multiple versions of a file in terms of a specified structure and to report changes relative to some parent structure. Our tool is unique because we seek to build a tool that can do a structural comparison of files in general. In fact, we could use our `xudiff` to compare Cisco IOS configuration files as well as C source code, Troff, JSON, or XML vocabularies.

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 [13]. Unfortunately, the SmartDifferencer is proprietary. Finally, TkDiff [37], 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 [3]. 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 [7, 9]. Zhang and Shasha [48] 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" [36]. Other researchers have looked at techniques to compare CIM-XML for compliance [31], XML trees for version control [2], and Puppet network configuration files based upon their abstract syntax trees [40]. Recently, our poster that proposed xutools [42] was cited in the context of differential forensic analysis [15].

5.4 xutools in general

Our xutools operate on parse trees so that practitioners can process texts in terms of high-level language constructs.

Industry: Augeas [19] 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.

6 Future work

Our xutools set the stage for a variety of future research directions and tool development beyond the scope of this paper. First we discuss three ways in which we want to improve and extend our xutools. We then will describe two second-order services built on top of our xutools to demonstrate the potential broader impact of our research.

6.1 Improvements and extensions

First, we want to extend the library of grammars on which our current xutools operate. In Section 4 we discussed three strategies to build up the grammar library for standard languages. For non-standard languages we want to consider how one might construct a grammar to recognize strings *similar* to a patch. As discussed in Section 2

this would allow us to extract code that is similar to a known vulnerability.

Second, we want to collect more feedback on our currently-implemented xutools and then re-implement them in C with a more efficient parser. Once this is done, we can benchmark our xutools against traditional Unix tools.

Third, we want to think about extensions to current xutools, new xutools, and how both interact with traditional Unix. As mentioned earlier, we could think about `head` and `tail` as extensions of `xugrep` in which we only report structures within a given range of nodes on our parse trees. Traditional `grep(1)` has an `--invert-match` option, the analogue of which in `xugrep` would also be quite useful. Other practitioners have suggested that a context-free `sed(1)` might be useful. We also have been thinking about how Unix pipelines and existing Unix tools interoperate with our xutools. Our `-l` option in `xugrep` for example, allows us to map newline-escaped strings in context-free languages to a set of lines that other Unix tools can use.

6.2 Second-order xutools services

Even if a system administrator has no interest in working directly with XML, C, Java, or some other configuration format on the command line, our xutools can still provide useful functionality. We now provide two examples to illustrate our claim.

A Structural Difference Engine for Version-Control Systems: 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. In essence, `xudiff` gives us an easy way to create a heatmap for the most volatile regions of configuration files and this is a useful tool for debugging. In LISA 2011, Workshop 8: Teaching System Administration stated that the parts of the system that break down historically are the first places to look for bugs. Our tools would allow practitioners to pinpoint the regions of configuration that were changed when the bugs first were reported. We also think such an application of our tools would help organizations understand legacy networks acquired via mergers.[6]

Measuring the Evolution of Terms-of-Service Policies: An important part of evaluating a web-based service is to understand its terms and conditions. These terms and conditions are described in a service's terms of service agreement. Casual readers, however, often do not read these terms, but rather agree by using the service. Despite this, enterprises and individual consumers need to be able to compare terms of service so that they can reliably evaluate the risks of using a service. The *Electronic Frontier Foundation (EFF)* recognizes the importance of understanding changes to security policies and so built a terms of service tracker, the TOSBack [14].

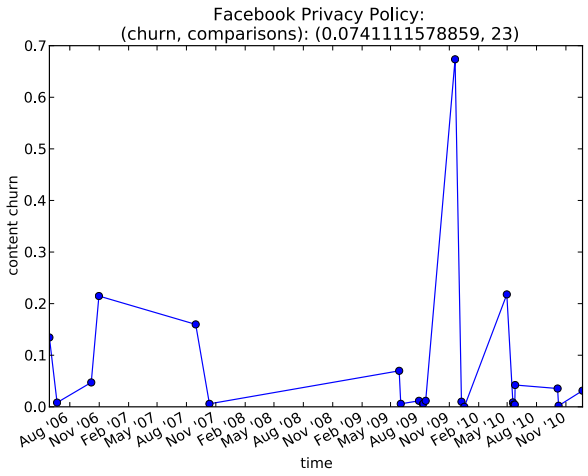

Figure 4: The structural evolution of Facebook Privacy Policies recorded by the EFF TOSBack from May 2006 until December 2010. We notice a spike in November 2009 when Facebook introduced major changes to their privacy policy that received enough attention to merit a subsection in Wikipedia's article: Criticism of Facebook [12].

As a proof-of-concept, we downloaded 24 Facebook policies recorded by EFF's TOSBack between May 5, 2006 and December 23, 2010. Using a simple metric based upon the Zhang-Shasha tree edit distance [48], we were able to quantify and thereby visualize the evolution of the Facebook privacy policy. In Figure 4, we notice a that our change metric (content churn) reaches a maximum on November 19, 2009. This policy revision was significant enough to merit an entire subsection in the following Wikipedia article: Criticism of Facebook. The article states that in November 2009, Facebook proposed a new privacy policy and new controls. These changes were protested and even caused the Office of the Privacy Commissioner of Canada to launch an investigation into Facebook's privacy policies [12]. This initial pilot study suggests that our xutools-based change metrics seem to be able to pinpoint important events in the "big-picture" history of a terms-of-service policy. Moreover, we can use our `xudiff` tool to "drill-down" and find specific policy changes.

7 Conclusions

Structured-text file formats break many of the assumptions upon which Unix text-processing tools were based. We have designed and built xutools, specifically `xugrep`, `xuwc`, and `xudiff`, to process texts in language-specific constructs. *In effect, we have extended the class of languages upon which Unix text-processing tools operate*. These generalizations are directly motivated by real-world problems faced by network and system administrators, software engineers, and other IT professionals who demand tools to process structured-text file formats.

8 Acknowledgments

The authors would like to thank Doug McIlroy for his encyclopaedic knowledge about Unix, his encouragement and advice. We would like to thank Tom Cormen, for his encyclopaedic knowledge about algorithms and insisting on a more rigorous mathematical model for our work. We would like to thank Rakesh Bobba and Edmond Rogers for their work to apply these tools in the domain of the power grid. We would like to thank Sergey Bratus for his insights into workflow patterns in Unix. We would also like to thank Kurt Seifried, Paul Schmidt, and many other practitioners for their real-world feedback. Finally, we would like to thank our LISA shepherd, Mike Ciavarella. This work was supported in part by the TCIPG project from the DOE (under grant DE-OE0000097). Views are the authors' alone.

9 Availability

Our xutools are available under the GPLv3 at `http://www.xutools.net/`. Our repository contains several use cases as well as relevant literature suggested by those we met as a result of our poster presentation at LISA 2011.[7]

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1988.

[2] 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.

[3] Bille, P. A survey on tree edit distance and related problems. *Theoretical Computer Science 337*, unknown (June 2005), unknown.

[4] 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.

[5] Burnard, L., and Bauman, S. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, 5 ed., 2007.

[6] 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.

[7] 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.

[8] 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`.

[9] 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.

[10] Coccinelle: A program matching and transformation tool for systems code, 2011. Retrieved November 11, 2011 from `http://coccinelle.lip6.fr/`.

[11] Codeanalyzer. Retrieved May 17, 2012 from `http://sourceforge.net/projects/codeanalyze-gpl/`.

[12] Criticism of Facebook. *Wikipedia: The Free Encyclopedia* (2012). Retrieved September 19, 2012 from `http://en.wikipedia.org/wiki/Criticism_of_Facebook#November.2FDecember_2009`.

[13] Designs, S. Semantic designs: Smart differencer tool. Retrieved May 16, 2012 from `http://www.semdesigns.com/Products/SmartDifferencer/`.

[14] TOSBack — the Terms-Of-Service Tracker. Retrieved September 19, 2012 from `http://www.tosback.org/`.

[15] 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.

[16] Grunschlag, Z. cfgrep - context free grammar egrep variant, 2011. Retrieved November 11, 2011 from `http://www.cs.columbia.edu/~zeph/software/cfgrep/`.

[17] Grushinskiy, M. XMLStarlet command line XML toolkit, 2002. Retrieved May 15, 2012 from `http://xmlstar.sourceforge.net/`.

[18] Jaakkola, J., and Kilpelainen, P. Using sgrep for querying structured text files. In *Proceedings of SGML Finland 1996* (October 1996), unknown, p. unknown.

[19] LUTTERKORT, D. Augeas–a configuration API. In *Proceedings of the Linux Symposium* (July 2008), pp. 47–56.

[20] 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.

[21] MCGUIRE, P. pyparsing, 2012. Retrieved September 19, 2012 from http://pyparsing.wikispaces.com.

[22] Metrics 1.3.6. Retrieved May 17, 2012 from http://metrics.sourceforge.net/.

[23] 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.

[24] PARR, T. ANTLR parser generator. Retrieved May 17, 2012 from http://www.antlr.org/.

[25] PIKE, R. Structural regular expressions.

[26] 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, p. unknown.

[27] Windows PowerShell. Retrieved February 3, 2012 from http://technet.microsoft.com/en-us/library/bb978526.aspx.

[28] RANCID - Really Awesone New Cisco Config Differ, 2010. Retrieved December 1, 2010 from http://www.shrubbery.net/rancid/.

[29] RELAX NG home page. Retrieved May 17, 2012 from http://www.relaxng.org/.

[30] 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.

[31] 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.

[32] RUBIN, P., AND MACKENZIE, D. *wc(1) Manual Page*, October 2004. Retrieved May 16, 2012.

[33] SIPSER, M. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, Massachusetts, 2006.

[34] 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.

[35] 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.

[36] 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.

[37] TkDiff. Retrieved February 3, 2012 from http://tkdiff.sourceforge.net/.

[38] UNKNOWN. *diff(1) Manual Page*, September 1993. Retrieved May 17, 2012.

[39] UNKNOWN. *grep(1) Manual Page*, January 2002. Retrieved May 17, 2012.

[40] 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.

[41] Vil. Retrieved May 17, 2012 from http://www.1bot.com/.

[42] 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.

[43] xmllint. Retrieved May 16, 2012 from http://xmlsoft.org/xmllint.html.

[44] W3C XML Schema. Retrieved May 17, 2012 from http://www.w3.org/XML/Schema.

[45] XML Path language (XPath), 1999. Retrieved May 15, 2012 from http://www.w3.org/TR/xpath/.

[46] The LEX and YACC Page. Retrieved May 17, 2012 from http://dinosaur.compilertools.net/.

[47] ZAZUETA, J. Micro XPath grammar translation into ANTLR. Retrieved May 16, 2012 from http://www.antlr.org/grammar/1210113624040/MicroXPath.g.

[48] 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.

Notes

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

[2] Thanks to Tanktalus for the Slashdot post.

[3] Our qualitative evaluation does not include performance benchmarks against traditional Unix tools, we see this as future work for after we get practitioner feedback on our tools and re-implement them in C. Once we have implemented them in C, we can do a fair comparison between our xutools operating on files in terms of lines and traditional Unix tools.

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

[5] In fact, the notion of managing and measuring changes to multi-versioned texts over time is based upon our previous work in the Classics at Holy Cross, Harvard's Center for Hellenic Studies, and the Perseus Project at Tufts University.

[6] According to Doug McIlroy, this idea is reminiscent of a visualization of change activity for switching code at Bell Labs. This visualization was possible because every change was manually registered in their source code control system. `xudiff` enables one to summarize such changes directly from the source.

[7] We received a lot of interest in our LISA 2011 poster [42], including worldwide press coverage of our work. See http://www.cs.dartmouth.edu/~gweave01/grepDiff/Press.html for details.