GCNav - Generic Configuration Navigation System

Shankaranarayanan Puzhavakath Narayanan
Purdue University
spuzhava@purdue.edu

Seungjoon Lee
AT&T Research
slee@research.att.com

Subhabrata Sen
AT&T Research
sen@research.att.com

***Abstract*—Configuration navigation and change-auditing is one of the most complex yet common tasks performed by network operators on a regular basis. Change-auditing router configuration files accurately is a challenging task due to presence of structure and hierarchy in the config content. Generic diff tools do not have the notion of context or syntactic structure while comparing files and produce diff reports (using minimum edit distance) that often do not match operator expectations. Moreover, these tools perform redundant (and expensive) comparison operations across contextually unrelated sections of the config file which makes them scale poorly even for config files of moderate size. On the other hand, vendor specific and customized diff solutions are not generic enough to be applied uniformly across a heterogeneous network. Also, modeling the configuration semantics for different vendors is a non-trivial and expensive process.**

In this paper, we introduce *GCNav*, a system that helps network operators perform general or customized change-auditing at varying levels of granularity on the network. Unlike existing solutions, *GCNav* makes use of the inherent syntactic structure common to all config files and thereby remains generic without compromising on the accuracy of results. Our experience with the deployment of *GCNav* on a large operational customer-facing IP network shows that it is able to provide a generic, accurate and scalable solution for change-auditing router config files. Our results show that *GCNav*'s diff results matches operator expectation while generic diff tools reported at least some misleading diff in 95% of the files analyzed. We also find that *GCNav* performs 7 times faster than customized auditing tools making it a feasible solution for online and interactive config auditing.

I. Introduction

Operators who manage large networks are required to perform a wide range of evolutionary, operational and maintenance activities which are often realized through modifications to the router config files. These activities usually cause a flurry of changes to a number of router config files consisting of tens of thousands of lines of arcane, complex, low level vendor-specific code that require a high degree of domain knowledge about the structure and vendor-specific details to understand and modify. Any errors in such operational activities can have serious adverse implications for the operation of the network ranging from immediate network outages and SLA violations to "hidden time bomb" latent misconfiguration [18], [10], [19].

Operators usually perform static analysis of their network device config files (called config auditing) to minimize and troubleshoot misconfiguration-induced outages. In this paper, we focus on one of the basic static analysis aspects: finding the differences between two configs. This is a key task that network operators perform in outage scenarios (e.g., identify config changes that occurred prior to the outage). Also, operators apply this capability to *template auditing* [4] to ensure router configs have all the intended configuration policies (captured in a *template*).

While conceptually simple, computing the *diff* of config files is challenging due to structure and hierarchy of content in these files, as well as the need to accurately identify the context of a change. Operators need to understand precisely where in the hierarchy the changes have occurred (e.g., for rapid understanding of the context of a change, or because changes high up in the config hierarchy might have a larger impact on the configuration). Today operators typically resort to a laborious, manual-intensive and error-prone approach involving a combination of "stare and compare" and the use of UNIX *diff* tools (like *GNUDiff* [11]) for comparing two config files. However generic tools like *GNUDiff* are poorly suited to the task as they compute the minimum edit distance between the two config files and do not have any notion of structure or hierarchy. As a result these tools often produce misleading diff results (see Section V-B2). Similar issues are also observed with the generic *diff* tools that operate on structured documents (like XML [23], [8]). While XML helps capture the syntactic structure of config files, existing XML diff tools compute minimal edit distance which is inaccurate for our purpose as we shall see in Section V-B. Also, these generic tools perform certain computationally expensive content-match operations (that are redundant when comparing config files) making them scale poorly for even small config files as we observe in Section V-C.

There are two broad approaches for a more tailored static analysis of config files. The first involves using vendor-specific solutions like [5]. However, these solutions face several challenges like high development cost, lack of flexibility and availability for devices from multiple vendors and different config languages. The second approach is to convert the various config languages to a common model and compare them. However, such extensive modeling of device configs is an extremely expensive process. It is also non-trivial to design a generic model that captures enough details for subtle differences in different config languages, especially when vendors constantly come up with different languages (i.e., OS upgrades) and new features for enhanced services (e.g., new policy statements).

In this paper, we present *GCNav*, a flexible config auditing system that is accurate yet generic enough to be applicable to a wide range of config languages and vendor types. Unlike

```
Interface ethernet 0/0
  description “loopback for server 1”
  ip-address 10.1.1.1 255.255.255.255
  snmp ifindex persist
  shutdown

Interface ethernet 0/1
  description “loopback for server 2”
  ip-address 10.1.1.2 255.255.255.255
  snmp ifindex persist
```

(a) Day X

```
Interface ethernet 0/0
  description “loopback for server 1”
  ip-address 10.1.1.4 255.255.255.255
  snmp ifindex persist
  shutdown

Interface ethernet 0/1
  description “loopback for server 2”
  ip-address 10.1.1.5 255.255.255.255
  snmp ifindex persist
  shutdown
```

(b) Day Y

```
3c4
<    ip-address 10.1.1.1 255.255.255.255
---------
>    ip-address 10.1.1.4 255.255.255.255
9c10
<    ip-address 10.1.1.2 255.255.255.255
---------
>    ip-address 10.1.1.5 255.255.255.255

11a12
>    shutdown
```

(c) GNUDiff output

Fig. 1. *GNUDiff* does not provide sufficient context when a router config changes.

```
<Interface >
   <Naming>
      <Type> ethernet </Type>
      <ID>0/0 </ID>
   </Naming>
   <description> loopback for server 1 </description>
   <ip-address >
      <ip>10.1.1.1 </ip>
      <msk>255.255.255.255 </msk>
   </ip-address>
   <snmp>ifindex persist</snmp>
   <shutdown/>
</Interface>
```

Fig. 2. XML config corresponding to interface 0/0 from Figure 1(a)

existing solutions, *GCNav* makes use of the inherent syntactic structure in the config files that is not only common to a majority of routers (and vendor types) but also independent of the functionality that is associated with the specific configuration logic. We designed *GCNav* to work with XML config files based on the following key observations. (i) XML has rapidly become the de-facto standard for representing any hierarchical information that has consistent structure. (ii) All major vendors in the networking industry have adopted XML config files [6], [12]. (iii) Large ISPs have already built parsers (e.g. [3]) to convert the CLI config files to XML config files.

To summarize, our contribution in this paper is *GCNav*, a configuration navigation system that provides accurate and contextual change-audit reports for router config files. We design *GCNav* with the goal of achieving the following key design goals:

- Developing a generic and scalable structured-*diff* methodology based on the syntactic and structural properties of config files that is independent of the semantics of the config language.
- Building on the existing XML *diff* techniques and adapt them to leverage the key structural properties exhibited by config files.

Our evaluation of *GCNav* on a large operational customer-facing enterprise IP network (anonymized as *IPNet*) show that it provides accurate *diff* reports. In comparison, more than 20% of the diff reported by *GNUDiff* was misleading in more than half the config files in the network that had reported some change. Our evaluation also shows that *GCNav* is 7 times faster than GSAT[4] (a domain specific tool developed for template auditing) while performing the same audit task.

II. Background and Problem Motivation

In this section, we discuss the nature of router config files and the structural properties exhibited by them. We then discuss the limited utility of existing *diff* tools for generating accurate *diff* on hierarchical content.

A. Nature of config files

We first illustrate some key properties of router config files, as noted in a previous work [21]. Figure 1(a) shows a section of the anonymized CLI (Command Line Interface e.g., [1]) code from the config files of an Alcatel router deployed in the *IPNet* network. This CLI code configures two ethernet interfaces on the router along with its properties like description, IP address and mask, an option for SNMP (Simple Network Management Protocol), and interface status (i.e., enabled or not). We make the following important observations from the example.

Low-level complex code: Router config files consist of thousands of lines of complex low-level commands. While we present a straightforward example here for simplicity, in reality, it is highly challenging to comprehend the intent and impact of these config code without sufficient domain knowledge and context information.

Structure and hierarchy: Router config files have some inherent syntactic structure and hierarchy enforced by the configuration logic. For example, in Figure 1, the CLI code can be grouped into logical blocks and sub-blocks such as $interface$, $ip-address$ or $description$. Also the $interface$ block encloses the $ip-address$ and $description$ blocks. This structural nature of config files is common to all devices, vendor types and is completely independent of the semantics of the config logic. *GCNav* takes advantage of these properties to gain accuracy and provide contextual information when comparing two config files.

Modeling configs as trees: A key abstraction that we use in *GCNav* is to utilize syntactic structure of config files and view them as trees [4]. Figure 3 shows an example of two simple config snippets modeled as trees where each config tree has four interfaces and for simplicity, we show only two properties (i.e., description and IP nodes) for each interface node. While modeling config files as trees help capture the structure without having to understand semantics, the next section explains why computing config *diff* is still a challenging task with existing *diff* tools.

B. Limited utility of existing diff tools

In this section we discuss the existing generic *diff* tools that broadly fall into two categories, text and XML *diff* tools.

1) Text Diff tools: We first focus on *GNUDiff* [11] which we shall use extensively for comparison in the rest of this paper. Though *GNUDiff* is very different to *GCNav* in its design and operation, we find that it was the most frequently used diff solution by the network operators due to familiarity and simplicity of the tool. *GNUDiff* is widely available and can be used to *diff* any two text files making it the de-facto *diff* tool of choice. We use the comparison between *GCNav* and *GNUDiff* to highlight the lack of accuracy with generic *diff* tools and emphasize the need for a novel approach.

Consider the *GNUDiff* output for the config snapshots shown in Figure 1. Here, *GNUDiff* identifies the changes

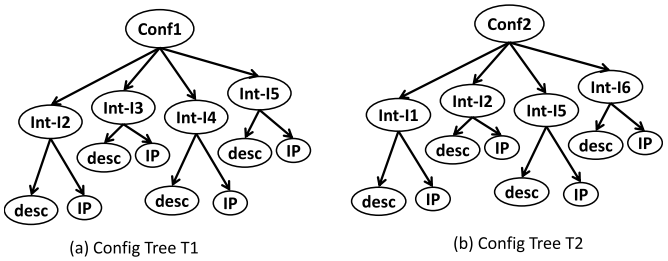


Fig. 3. Comparing config trees

between the two configs (i.e., IP address change in both interfaces and status change in the second interface), but does not associate the change with the corresponding *interface*. This is because *GNUDiff* has no notion of the syntactic structure of the router config and is therefore unable to provide sufficient context to detected changes. While making this association is simple in our small example, as config files become longer (e.g., thousands of lines), this information becomes critical for network operators to understand the implications of the change. Also, *GNUDiff* reports are often misleading since computing the minimal edit distance can potentially match equivalent but contextually unrelated blocks(see Section V-B).

2) XML diff tools: We shall now see why the generic XML *diff* tools are not suitable for the static analysis of XML config files. Figure 2 shows the XML config snippet corresponding to Interface $0/0$ in CLI sample from Figure 1(a). While XML helps capture the syntactic structure of config files, the problem of identifying the *diff* presents many challenges. Existing XML and structured diff tools (E.g., [23], [8]) compute minimal edit distance which is inaccurate for our purpose. Moreover, these tools try to perform expensive operations like matching sub-blocks of config code in an entire config file which does not scale even for small config files (see Section V-C). Therefore, we design *GCNav*'s algorithms based on our insights on the syntactic and structural characteristics of config files to generate accurate reports in an efficient manner.

III. *GCNav* Architecture

Our design of *GCNav* takes advantage of the following properties exhibited by config files.

1) Blocks (nodes in config tree) of config code do not move to arbitrary sections of the config file. Consider the config trees in Figure 3. The *ip* nodes would be a child of some *interface* node but would not be the child of any other arbitrary node such as *desc* or *conf*, as certain config commands are allowed only in relevant contexts.
2) Two nodes that look similar but have a different *signature* (path to the root) can represent contextually different parts of the config code. For example, an *ip* node attached to *Int-I2* is contextually different from the *ip* node that is attached to *Int-I3*), even if the two *ip* blocks have the same content. This is because they configure two different interfaces and comparing them would result in a *diff* that misleads the network operators.

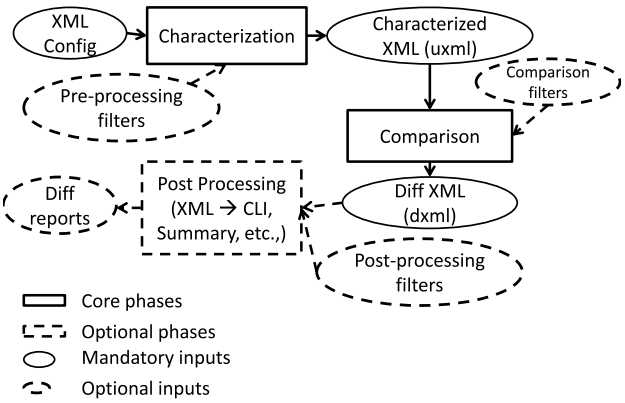


Fig. 4. Architecture of *GCNav*.

We derive these key insights from configuration logic and syntax of config files that are common to all routers and vendor types. Since potential matches occur only at nodes that have similar *signatures*, *GCNav* only compares the nodes at the same depth in the config tree. Conventional tree *diff* algorithms [7] typically tend to match the above mentioned cases, which is both incorrect and expensive. On the other hand, *GCNav* benefits from these insights by pruning parts of the tree and improving the scalability of the system.

Figure 4 shows the block diagram of the architecture of *GCNav*. We first present the two core phases that are performed on the XML config files in the following sequence to obtain the *diff*: (1) Characterization of the XML files and (2) Comparing the XML files.

A. Characterization

In order to accurately compare any two config files(trees) $T1$ and $T2$, we need to efficiently (1) identify whether for each node $T1$, there exists a corresponding node in $T2$ to compare against and (2) determine if the two corresponding nodes are identical or different. For this purpose, we annotate every node of the XML config file with two special attributes (internal to *GCNav*) in the *Characterization* phase. The annotated XML config file produced by this step (called a *characterized* XML file) is used as input to the comparison phase. We further elaborate on the above two points in this section.

1) Identifying corresponding nodes: Suppose we want to compare two config trees $T1$ and $T2$ shown in Figure 3. The *diff* process should be able to associate the node *Int-I2* from $T1$ with its corresponding node *Int-I2* in $T2$. Brute-force approaches like pair-wise comparison (e.g. every child of *Conf1* in $T1$ with every child of *Conf2* in $T2$) can considerably slow down the *diff* process. Therefore, we introduce a new attribute called *UID* (Unique Identifier) to every node in a config tree which is used to identify the corresponding nodes in the comparison phase. For the ease of exposition, we show a simple *UID* (e.g., *I2*) for each interface node in Figure 3, while we use a MD5 hash in practice.

Computing the *UID*: If a node in a config file has identifying attributes such as *ID* or *NAME*, then a simple solution for computing the *UID* is to use those attributes. However, node attributes are not always mandatory. The content of child

nodes can sometimes be used as identifiers (e.g., interface name or IP address for an interface node). This approach necessitates detailed knowledge of config semantics, which should be avoided to make *GCNav* generic and flexible.

GCNav employs heuristics based on the structural properties of config files to automatically identify nodes that can be used as *UID*. To ensure correctness, the heuristic needs to identify at least one node that can be used to compute the *UID*. A simple heuristic is based on the observation that leaf nodes are often good candidates for *UID* (e.g., *description*, *ip*). A stronger heuristic that worked very well for our network is to use the *HashValue* of non-repeating children to compute the *UID* for a given node. This heuristic would identify the nodes like *Naming* and *description* (in Figure 2) that are used as identifiers even in their semantic model. Apart from implementing these heuristics, *GCNav* also provides the operators the option to explicitly suggest an *UID* as an external specification to the system.

2) Summary information for node content: Once the corresponding nodes from given two trees are identified, we need to determine whether the two nodes are equivalent in its content (sub-tree) and value (attributes). A simple but expensive solution is to traverse the two subtrees recursively and check the equivalence at every level in the tree. To avoid the expensive tree traversals during the comparison phase, we calculate a simple MD5 hash on an entire subtree and annotate the root of the subtree with the hash value. Thus, the equality of content can be quickly determined for the given two nodes by comparing the hash values. The *HashValue* can be efficiently computed by performing a bottom-up traversal of the config tree and propagating the *HashValue* of a node to its immediate parent node. Computing the *UID* and *HashValue* attributes is independent of the *diff* process and needs to be done exactly once for each config file. Therefore, *GCNav* performs the characterization as a part of the pre-processing phase that can be delegated as an offline process.

Putting it together: At a high level, the characterization algorithm computes and adds the *HashValue* and *UID* (*computeHash()* and *computeUID()*) for every node in the config tree using the standard bottom up tree traversal. The annotated XML file produced by the characterization phase is called the *uxml* file.

B. Comparison

The *comparison* phase is undoubtedly the most important component of *GCNav*'s architecture which performs the actual *diff* between the two config files that need to be audited. The *comparison* phase takes the *characterized* files as its input and executes the diff algorithm to produce the *diff* result as a set of modifications, additions and deletions. It is important to note that the *comparison* phase is generic and totally independent of the type or vendor of the config files. The output of the *comparison* phase is an annotated XML file (*dxml* file) with attributes added to each node indicating the *added*, *deleted*, *modified* or *unchanged* status of the node.

Algorithm 1 Compare

```
    procedure COMPARE(t1, t2)
        if t1.root.hash == t2.root.hash then return No diff
        while ∀ node in BFS of t1, t2 do
            if node marked for process then
 5:             (n1, n2) ← (nextNode(t1), nextNode(t2))
                (C1, C2) ← (n1.children, n2.children)
                for each c1 in C1 do
                    if c2 = match(c1, C2) then
                        if c1.hash == c2.hash then
10:                         c1, c2 equal
                            for c upto c2 in C2 do
                                markAdded(c)
                        else
                            markModified(c1, c2)
15:                 else
                        markDeleted(c1)
                if ∀c in C1 seen then
                    markAdded(Nodes remaining in C2)
                markDeleted(Nodes remaining in C1
20: end procedure
```

At a high level, the comparison algorithm performs a BFS (Breadth First Search) tree traversal on both the config trees simultaneously. Based on our observations described earlier in this section, the algorithm looks for a potential match only between nodes that have a consistent *signature* (path from the root). Among those nodes with the same signature and depth, the algorithm identifies the corresponding node pairs using *UID* and determines the equivalence using *HashValue*.

The following are the possible cases after a comparison:

1) Nodes have equivalent *UID* and *HashValue* and do not have a *diff*. Our construction of the *HashValue* ensures that the entire subtrees are the same and therefore further traversal down the subtrees are redundant.
2) There exist corresponding node pairs with the same signature, depth, and *UID*, but they have different *HashValue* values. This indicates a *modification* (*mod*) in the subtree and we further process the subtrees according to the BFS traversal order.
3) A node that has no corresponding node with a matching signature, depth, or *UID*. This indicates an *addition* or *deletion* of the node based on the config tree where the node is located.

Algorithm 1 shows the pseudo-code of the algorithm used in the comparison phase of *GCNav*.

C. Optional Phases

The optional phases of *GCNav* are the pre-processing and post-processing phases as shown in the Figure 4.

Pre-processing XML files: Large enterprise or ISP networks are highly heterogeneous consisting of config files that may have some redundant or irrelevant config blocks like timestamps, version number, device information etc. Diff computed on such redundant config blocks often prove to be distracting for the operators. Also evolving networks typically contain legacy devices that require some special handling before they can be processed. To address this requirement, we designed the characterization phase to optionally accept a set of pre-processing filters (using our filter language described below)

as an external input from the operators. We discuss some of these aspects in detail in Section IV.

Post-processing XML files: As mentioned in Section II-A, config files are large, complex and hard for human users to read and understand. It is therefore important for *GCNav* to have the ability to focus on certain sections of the config file that are important for the operators. For instance, network operators might want to view changes to the *interface* blocks alone and ignore any changes to the *description* blocks. To accommodate these requirements, we provide *GCNav* with a simple and intuitive filter language which uses *include* and *exclude* constructs to accept filtering queries from the operators as XPath[24] queries. A sample query expression for our example from Figure 2 would look like *i-//interface:e-//description*, where multiple filter queries are separated by colon. The first half of the expression indicates whether it is an (i)nclude or (e)xclude construct and the second half of the expression gives the actual XPath query to be evaluated. For the above query, the filtering component would show *diff* sections that include *interface* nodes and excludes the *description* nodes from the results.

It is important to note that the pre and post processing phases are optional features that are provided to the operators to fine tune the system. These phases do not compromise the generality of our system and accepts a set of filter rules as its input and generates the *diff* reports accordingly. As a final note, *GCNav* is written completely in Python and uses *lxml* [17] for its XML library support.

IV. Practical Considerations

In this section, we briefly discuss a few practical aspects that we encountered and needed to resolve while deploying *GCNav* on large ISP networks consisting of heterogeneous devices. This discussion, in particular, highlights the benefit of designing *GCNav* as a set of independent phases where the core phases remains well insulated from the quirks of the heterogeneous network.

Legacy devices only with CLI config files: Some legacy devices may not support XML config files. In such cases, *GCNav* can piggy-back on tools that can convert the CLI configs into structured XML configs [3]. For completeness, we include these solutions as a part of the pre-processing phase.

Errors in the XML files: From our initial experiments, we found that some config files collected from devices contained malformed XML constructs. These could have been due to manual configuration errors or even more systematic issues such as missing root nodes in the XML files. These issues can be ignored or fixed by custom or automated scripts in the pre-processing phase.

Vendor and device specific oddities: XML config files may have odd constructs that can mislead the comparison phase. For e.g., a certain config file that we processed had a timestamp attribute at the root node (wrapping the entire config file), which changes everyday. While this is a genuine modification in the config file, if we include that attribute in the *UID* and *HashValue* calculation (see Section III-A), the root node would be always considered modified, which clearly is not the desired outcome. For this purpose, in *GCNav*, we provide the ability to specify filters or pre-processing rules in the characterization phase, so that we do not include such attributes as timestamp in the *UID* and *HashValue* calculation. These pre-processing rules can be provided by the operators (who have the domain knowledge) as a custom specification to *GCNav* as a static metadata. Our experience with *GCNav* on *IPNet* shows that these oddities can be easily addressed using a few (less than 5) simple filter queries.

V. Evaluation

We evaluated *GCNav* on a large, operational, customer facing IP network consisting of thousands of routers. These routers have varied functionality and come from a wide range of prominent vendors like Alcatel, Cisco and Juniper. All our evaluations are performed on the XML config files of these routers obtained for a period of two months (January - February, 2011).

We compare *GCNav* with the two prominent classes of existing generic diff solutions that are commonly used (see Section II-B). We compare *GCNav* with (i) *GNUDiff* to highlight the lack of accuracy with the generic *diff* tools. (ii) GSAT [4] (a template auditing tool that uses structured XML diff) to show the benefit of *GCNav*'s insights in improving the performance while retaining its accuracy.

A. *Vendor neutrality of* GCNav

```
<interface ID ="1">
   <description>
      description1
   </description>
   <ip>
     <ip>
        address 11.111.11.111
     </ip>
     <ip>ip text2</ip>
   </ip>
   <service-policy>
      Service Policy1
   </service-policy>
</interface>
```

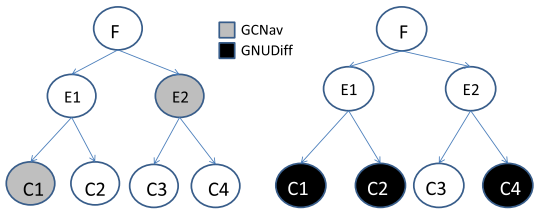
(a) Cisco config sample

```
<interface ID ="1">
   <address>
      address 11.111.11.111
   </address>
   <bfd>bfd text1 </bfd>
   <line>interface text2</line>
   <desc>
      description1
   </desc>
   <qos>qos3</qos>
</interface>
```

(b) Alcatel config sample

Fig. 5. Config snippet of an interface

Figure 5 shows an excerpt of the XML code used to configure an *interface* from the router config files of two prominent vendors, Cisco and Alcatel. It can be seen that though the code samples differ in syntax, they have very similar syntactic structure which is exploited by *GCNav*. In fact, the Alcatel code sample shown here was generated using a custom developed parser that generates XML from the CLI code. While we do not show here further details of configs from different vendors or different languages, we have extensively evaluated *GCNav* using router configurations from three major vendors - Alcatel, Cisco and Juniper. To emphasize this flexibility, in the following evaluations, we use config files from Alcatel devices in Section V-B and Cisco routers in Section V-C.



Node	F	E1	E2	C1	C2	C3	C4
GCNav	0	0	1	1	0	0	0
GNU Diff	0	0	0	1	1	0	1

Fig. 6. Measuring accuracy with symmetric difference

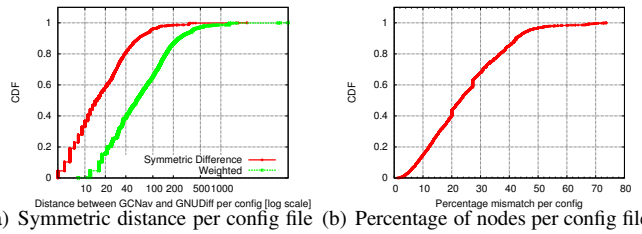
B. *Accuracy of* GCNav

In this section, we introduce a metric to measure the accuracy of *diff* tools when auditing router config files. We then evaluate the accuracy gained by *GCNav* by comparing it with the results produced by *GNUDiff* for over 3500 field config files from the *IPNet* network.

1) Measuring accuracy: The notion of accuracy while comparing config files is different from that of a traditional *diff*. A *diff* report is accurate if it is able to capture the context of the change reported. Consider the trees in Figure 3. If the node $I1$ is considered *added* to the tree $T2$, then children of $I1$ ($desc$ and ip) should also be considered implicitly as *added*. In other words, the impact of a change high up in the tree should be reflected in the entire sub-tree.

Since there is no standard measure of correctness of config *diff* reports, we verified *GCNav*'s accuracy based on operator expectations and feedback. However, we measure the accuracy gained by *GCNav* over *GNUDiff* by introducing a distance metric which represents the amount of dissimilarity between the *diff* results of the two tools. Since *GNUDiff* computes the diff by treating the XML config as flat file, we map its result back to the XML tree to ensure a fair measurement of the distance. Figure 6 shows the diff trees generated using the two *diff* methods. Intuitively, accuracy gained can be defined as the amount of dissimilarity between the two trees which can be computed using the symmetric distance (number of nodes which disagree in their results). The table in Figure 6 shows how a simple symmetric distance is computed for the *diff* trees shown above it. Further, to capture the effect of hierarchy, we associate weights based on the depth of the node in the config tree while computing the symmetric difference. The root should have the highest weight value (equal to the height of the tree) since it indicates a change in the entire config file, while the leaf would have the least value (a weight of one).

2) Comparing accuracy of GCNav *and* GNUDiff*:* We evaluated the accuracy gained by *GCNav* over *GNUDiff* using the weighted symmetric difference scheme mentioned above. Figure 7(a) shows the CDF of the symmetric distance and weighted distance between the results of *GCNav* and *GNUDiff* for about 3348 routers that had been modified across a period of 7 days. The graph shows that *GCNav* and *GNUDiff* differ at 20 nodes or more for 40% of the config files (where at least one of the methods reported a change). An even more surprising observation is that among the routers that had some change reported, less than 5% of the routers agreed completely on their *diff* results from the two tools. Figure 7(b) shows the CDF of the percentage of mismatch per config file between the *diff* reports of the two tools. It can be observed from the graph that about 40% of the config files that reported some change had more than 25% mismatch between the two methods. Our results show that traditional *diff* tools are very poorly suited for change-auditing config files. They also highlight the need for a system like *GCNav* which can provide *diff* reports that match operator expectations.



(a) Symmetric distance per config file between the two diff reports

(b) Percentage of nodes per config file that do not match across the two diff reports

Fig. 7. Comparing accuracy of *GCNav* with *GNUDiff*

We now explain with an example why *GNUDiff* was inaccurate when comparing the config files. Consider Figure 8 showing an excerpt from a XML field config file for two days (a) and (b). Figure 9 shows the comparison of the *GNUDiff* output (a) with the ideal (expected) output (b). *GNUDiff* is very different from the ideal diff result due to two reasons. First, it has no notion of structure in the config file and hence it is not able to associate a change at a node to its sub-tree. Second, it tries to compute the minimal edit distance which can lead to incorrect comparison. For instance, it matches node $e1$ from (a) with node $e2$ from (b) as they have a maximal match and reports the sub-tree of $e1$ from (a) as added. This phenomenon skews the *GNUDiff* output from the expectation as observed in the example.

C. *Response Time of* GCNav

We evaluated the performance of *GCNav* by comparing it with a template auditing tool called GSAT [4] where the goal

```
<e ID ="1">
 <action>a1</action>
 <description>d1</description>
 <line>e1 create</line>
 <match ID ="p1">
  <type>redirect</type>
    <line>match p1</line>
  </match>
</e>
```

(a) 17 Feb, 2011

```
<e ID ="1">
 <action>a3</action>
 <description>d3</description>
 <line>e1 create</line>
 <match ID ="p3">
   <type>redirect</type>
   <line>match p3 </line>
 </match>
</e>
<e ID ="2">
 <action>a1</action>
 <description>d1</description>
 <line>e2 create</line>
 <match ID ="p1">
   <type>redirect</type>
   <line>match p1</line>
</match>
</e>
```

(b) 24 Feb, 2011

Fig. 8. Config from two days with relevant blocks

```
Add:<action>a3</action>
    <description>d3</description>
     <line>e1 create</line>
     <match ID =“p3">
         <type>redirect</type>
         <line>match p3 </line>
     </match>
     </e>
     <e ID =“2">
Mod:<line>e1 create</line>
     <line>e2 create</line>
```

(a) GNUDiff

```
Del: <action>a1</action>
     <description>d1</description>
     <match ID =“p1">
Add:<action>a3</action>
     <description>d3</description>
        <match ID =“p3">
Add: <e ID =“2“>
```

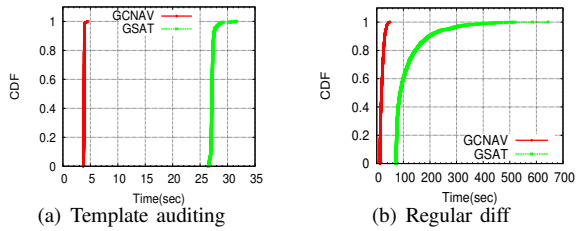
(b) Ideal

Fig. 9. Comparing *GNUDiff* with the ideal output

of *gold standard* auditing is to ensure all field configs(XML) are equivalent to a certified gold config(XML) provided by the operators. We use *GCNav*'s filter queries to produce reports that matched GSAT's audit reports (i.e., both reports have the same accuracy) for a fair comparison of their performance. Though seemingly similar, *GCNav* is very different from GSAT in its design and approach. GSAT was developed as a point solution for a specific problem while *GCNav* is meant to be a generic and flexible solution that can cater to a variety of audit requirements. Also, GSAT (which makes use of a generic perl module called *SemanticDiff*) is not scalable and therefore has limited practical use as a more general tool. In contrast, *GCNav* is scalable and can be used for any change-auditing requirements without compromising on accuracy.

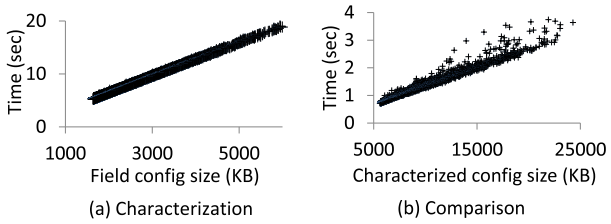
Figure 10(a) shows the CDF comparing the response time of *GCNav* and GSAT for computing the *diff* report for a set of 2500 XML config files in the network. The Y-axis shows the CDF and X-axis represents time taken in seconds to obtain the *diff* (all phases including the pre and post processing phases) between the corresponding field config and the gold config. From the graph, it can be seen that *GCNav* is 7 times faster than GSAT when computing the *diff*. Also the response time of *GCNav* is well within 5 seconds for all the config files making it a feasible tool for interactive auditing of router config files. On the other hand, GSAT takes more than 26 seconds for processing a *diff* making it too slow for interactive auditing. For this reason, GSAT was deployed as an offline tool.

We now evaluate the performance of *GCNav* and GSAT when comparing two versions of the same field config file across two successive days. Figure 10(b) shows the CDF comparing the response time of GSAT and *GCNav* to audit 2500 XML config files across two successive days. The Y-axis shows the CDF and X-axis represents time taken for computing the *diff* (all phases). The graph shows that *GCNav*



(a) Template auditing (b) Regular diff

Fig. 10. CDF comparing the Total response time of *GCNav* vs GSAT.



(a) Characterization (b) Comparison

Fig. 11. Time vs File size for the two key phases of *GCNav*.

performs at least 7 times better than GSAT. In particular, beyond the $50th$ percentile, GSAT shows exponential trends while the *GCNav*'s performance remains steady. Also, *GCNav* has a very short tail (and $95th$ percentile less than 20 seconds) compared to that of the GSAT ($95th$ percentile greater than 300 seconds). Note that field configs are substantially larger in size than gold configs and therefore the problems with the scalability of GSAT becomes exacerbated. For this reason, GSAT was not practical for more general change-auditing requirements.

We compared the contribution of the various phases to the total time taken by *GCNav*. Our analysis showed that the *characterization* phase is the most expensive phase of *GCNav*, albeit taking less than 10 seconds for 60% of the 2500 files processed. A large of part of this time is spent on computing the *UID* (requiring traversal of the immediate child nodes) discussed in Section III-A. However, characterization needs to be done exactly once for a given config file and can be performed offline as soon a new snapshot of the config files arrive. It is important to note that the rest of the phases in *GCNav* takes less than 2 seconds for 90% of the files making it an effective tool even for online and interactive *diff* session.

We now perform a detailed evaluation on the scalability of *GCNav* by plotting the time taken by the characterization and comparison phases against different config file sizes. We make use of the XML field config files described in Section V-C for our evaluation of the scalability of *GCNav*.

Figure 11(a) shows the scatter plot representing the time taken by the characterization phase along the Y-axis and the input file size (field config) along the X-axis. Similarly, Figure 11(b) shows the scatter plot with the time taken by the comparison phase along the Y-axis and the input file size (characterized file) along the X-axis. From the graphs, we see that the time taken by both the phases increases linearly with the increase in input file size indicating good scalability of *GCNav*. Comparing the file sizes across the two graphs also reveals the characterization overhead. Characterized files are approximately thrice the size of the corresponding field configs due to the annotations that are added as a part of the characterization phase. *GCNav* aims to reduce the audit time by adding annotations to the config files and as a consequence increases the size of the intermediate files generated. However, should space requirements become critical, characterization can be limited to specific sub-trees of the config file using the filter language described earlier.

VI. Related Work

Change auditing is an important and frequent task performed in all enterprise networks. Many large enterprises have developed their own customized auditing solutions which is expensive and requires extensive modeling effort. Vendor specific solutions like [5] are another class of tools that have been used for auditing purposes. However, these solutions are either technology or vendor dependent and are not generic enough to be applicable for heterogeneous networks. *GCNav* on the other hand provides a generic methodology for change auditing in router configuration files. Configuration auditing has also been looked into from the perspective of solving problems like [20], [21], [13], [4]. Works like [16], [15] have looked at computing diff for Access Control Lists (ACLs) and firewall configuration. However, these are optimized and built to solve specific problems which often makes them slow or unsuitable to be used as a generic tool. On the other hand, *GCNav* adopts a more systematic and structured approach making use of the syntactic structure that is common to all router configuration files.

XML based router configuration has been explored in [22], and works like [7] have comprehensively studied the various algorithms that can be used for computing XML diff enumerating the features, advantages and disadvantages with each approach. Structured *diff* has been explored extensively in the field of version controlling [9], [2]. However, these solutions are again customized for versioning and optimized for merge operations making them unsuitable for auditing config files. Tools like XANADUE [14] have been used to detect changes in XML databases. However, unlike *GCNav* they are not context oriented, making them unsuitable for our purpose. *GCNav* differs from the generic XML *diff* tools [23], [8] in the following key design decisions. First, it does not compute the minimum edit distance which is fundamental to the accuracy of any config *diff*. Second, it does not match subtrees across different parents which is key to both correctness and scalability of the solution.

VII. Conclusions

In this paper, we presented *GCNav*, a configuration navigation system. Our design of *GCNav* focuses on a generic, accurate and scalable approach to change auditing in router configuration files. While existing solutions are customized towards solving specific problems, *GCNav* takes a first step towards developing a structured and general solution that retains the required accuracy and performance. *GCNav* uses the inherent syntactic structure common to all config files and thereby remains generic without compromising on the accuracy. *GCNav* takes advantage of key insights enforced by the syntactic structure of configuration files to optimize on performance of the algorithms, thereby making the system scalable. *GCNav*'s modular design renders it easily adaptable to a variety of audit requirements. Our evaluation shows that *GCNav* performs significantly better than the two prominent classes of generic diff solutions used by the operators. In the future, we also plan to compare the performance of *GCNav* with that of point solutions like semantically modeled or vendor specific tools.

GCNav has been an indispensable tool for change-auditing in a large operational customer-facing IP network for about a year and has catered to a variety of audit requirements. We believe that a system like *GCNav* which takes a general and structured approach to change-auditing in router configurations is a crucial first step that leads to an efficient and robust network management solution.

References

[1] The alcatel cli reference. http://enterprise.alcatel-lucent.com/docs/?id=12979.

[2] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured merge: rethinking merge in revision control systems. In *ESEC/FSE 2011*.

[3] D. Caldwell, S. Lee, and Y. Mandelbaum. Adaptive parsing of router configuration languages. In *INM 2008*.

[4] D. Caldwell, S. Lee, S. Sen, and J. Yates. Gold standard auditing for router configurations. In *LANMAN 2010*.

[5] Cisco contextual configuration diff utility. http://www.cisco.com/en/US/docs/ios/12_3t/12_3t4/feature/guide/gt_diff.html.

[6] Cisco ios xml reference. http://www.cisco.com/en/US/docs/ios_xr_sw/iosxr_r3.4/xml/program/guide/xl34cnfg.html.

[7] G. Cobéna, T. Abdessalem, and Y. Hinnach. A comparative study for xml change detection. *Research Report, INRIA Rocquencourt, France*, 2002.

[8] G. Cobena, S. Abiteboul, and A. Marian. Xydiff tools detecting changes in xml documents, 2002.

[9] S. Elmougy and W. Al-Adrousy. A structured-based differencing method for version control system for java codes. In *ISSPIT, 2010*.

[10] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. of NSDI*, May 2005.

[11] Gnu diff. http://www.gnu.org/software/diffutils/diffutils.html.

[12] Junos xml reference. http://www.juniper.net/techpubs/software/junos/junos94/swconfig-automation/advantages-of-using-the-junoscript-and-junos-xml-apis.html.

[13] F. Le, S. Lee, T. Wong, H. Kim, and D. Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM Trans. Netw.*, 2009.

[14] E. Leonardi and S. Bhowmick. Xanadue: a system for detecting changes to xml data in tree-unaware relational databases. In *SIGMOD 2007*.

[15] A. Liu. Firewall policy change-impact analysis. *ACM Transactions on Internet Technology (TOIT)*, 2012.

[16] A. Liu and M. Gouda. Diverse firewall design. *Parallel and Distributed Systems, IEEE Transactions on*, 2008.

[17] The lxml python toolkit. http://lxml.de/.

[18] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, Aug. 2002.

[19] S. Narain. Network configuration management via model finding. In *Proc. LISA*, Dec. 2005.

[20] Y. Sung, C. Lund, M. Lyn, S. Rao, and S. Sen. Modeling and understanding end-to-end class of service policies in operational networks. In *SIGCOMM 2009*.

[21] Y. Sung, S. Rao, S. Sen, and S. Leggett. Extracting network-wide correlated changes from longitudinal configuration data. *PAM*, 2009.

[22] L. Vanbever, G. Pardoen, and O. Bonaventure. Towards validated network configurations with ncguard. In *INM 2008*.

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

[24] Xpath query language. http://www.w3schools.com/xpath/default.asp.