V3SPA: A Visual Analysis, Exploration, and Diffing Tool for SELinux and SEAndroid Security Policies

Robert Gove
Invincea Labs

ABSTRACT
SELinux policies have enormous potential to enforce granular security policies, but the size and complexity of SELinux security policies make them challenging for security engineers and analysts to determine whether the implemented policy meets an organization’s security requirements. To address the challenges in developing and maintaining SELinux security policies, this paper presents V3SPA (Verification, Validation and Visualization of Security Policy Abstractions). V3SPA is a tool that can import source or compiled binary SELinux and SEAndroid policies. V3SPA has two main visualizations for imported policies: A policy explorer, and a policy differ. The policy exploration view is designed to support a policy analyst or designer in exploring a policy and understanding the relationships defined by the policy. The differ view is designed to support differential policy analysis, showing the changes between two versions of a policy.

The main contributions of this paper are 1) the design of the policy explorer, and the design and novel usecase for the policy differ, 2) a report on system design considerations to enable the graph visualizations to scale up to visualizing policies with tens of thousands of nodes and edges, and 3) a survey of five SELinux and SEAndroid policy developers and analysts. The results of the survey indicate a need for tools such as V3SPA to help policy workers understand the big picture of large, complex security policies.

1 INTRODUCTION
As system complexity has increased, security goals have become incomprehensible at the implementation and configuration levels of an operational environment. To secure modern enterprise systems, the ability for security professionals to define and visualize the overall security policies of the system is essential to understanding and verifying policy configurations. While some tools exist to implement the system security policies themselves, the complexity of operational systems clutter the ability of security engineers to map security goals to proper system configurations, thus leaving a high probability of security errors. Furthermore, as system complexity increases, so does the difficulty in assessing differential system modifications without requiring a ground up re-verification.

Many Linux distributions have adopted SELinux, including Fedora, Debian, Gentoo, and Ubuntu. SELinux has also been a part of Android since version 4.3. According to data collected in July, 2016, 81.1% of active Android devices are running a version of Android that includes SEAndroid. Considering the wide adoption of SELinux by Linux distributions, and considering that there are over one billion active Android devices in use, there is a clear need for policy designers to have appropriate tools to effectively design, implement, and verify SEAndroid policies. (In this paper, I use “SELinux security policies” to refer to both SELinux and SEAndroid security policies, unless stated otherwise.)

SELinux policies are challenging to understand and analyze due to their size and complexity. For example, running the search utility on the 20141203 version of the Tresys reference policy (the policy that all Linux distributions use as a base) shows there are 94,420 allow rules. These rules express the relationships between 3,410 subjects, 4,037 objects, 77 permissions, and 231 object classes.

This paper presents a novel system to address the challenges faced by security engineers and analysts. V3SPA is an interactive visualization and security policy analysis environment built on top of the Lobster domain-specific language [6].

To address the challenges faced by security engineers and analysts, this paper presents V3SPA, a novel tool for analyzing and exploring SELinux and SEAndroid security policies. V3SPA is designed to be a scalable system capable of visualizing all the allow rules of a policy simultaneously, but allowing users flexibility to selectively apply filters to see only the relevant components of the policy. V3SPA has two main visualizations: A policy explorer (Figure 1), and a policy differ (Figure 2). These visualizations are designed to explore a policy to better understand the rules and relationships in the policy, and to visually diff two versions of a policy and analyze the differences between them. I discuss V3SPA’s design, and I describe several design decisions and performance optimizations that enable V3SPA to scale up to visualizing tens of thousands of nodes and edges.

I show V3SPA’s utility by demonstrating V3SPA’s features while analyzing several SELinux and SEAndroid security policies. I then discuss the results of a survey of five SELinux policy developers. I compare V3SPA to the tools they commonly use, showing that V3SPA enables analysis that is difficult and time consuming using other tools.

2 SELINUX AND SEANDROID
Traditionally, Linux operated under a discretionary access control (DAC) model, where the owner of an object could grant access to it at the owner’s discretion. Security Enhanced Linux (SELinux) is an add-on for Linux that implements mandatory access control (MAC) based policies on top of the existing DAC model. In MAC, access control is specified at the system level. SELinux implements MAC through type enforcement rules in the security policy, where subject types (typically processes, also called domains) are granted permissions on object types (i.e. system resources such as files, sockets, shared memory, etc.) and object classes (the specific class of the object). In this system, nothing is permitted until it is specifically allowed by a policy rule. For example, the allow user_t bin_t : file read; rule says that the processes in the user_t domain are allowed to read the files of the bin_t type. SELinux security policies also include definitions for users, roles, and other information. This collection of rules is the SELinux security policy.

SEAndroid (Security Enhanced Android) adds a version of SELinux to the Android mobile operating system. SEAndroid policies have some differences to SELinux policies, but they are similar enough that many tools can be used to develop and analyze both types of policies.

For more information on SELinux, see Mayer et al [9].
Figure 1: The policy explorer in V3SPA. On the left, the editor pane shows controls for filtering the policy. On the right is a node-link diagram representation of the policy, where subjects are blue nodes, and objects and classes are paired together into a single orange node if they both occur in the same allow rule. A tooltip for the `sepgsql_client_type` subject node show statistics and filter options for the node.

Figure 2: The policy differ in V3SPA. On the left, the editor pane shows controls for filtering the nodes in the policies, which are divided into subjects, permissions, objects, and classes, and further divided based on which policy contains those policy components. On the right, the nodes are broken into four groups. Hovering over the `redis_log_t` subject node shows all the subjects, permissions, and classes that co-occur in the same allow rules as the `redis_log_t` subject.
3 RELATED WORK

Existing publicly available tools are commonly used in SELinux policy development and analysis. The Tresys SETools\(^2\) include several such command-line tools, such as sediff for comparing differences between two policies, sedta for performing domain transition analyses, seinfo for viewing the policy components, seinfoflows for analyzing information flow, and searsearch for searching the policy rules. SETools also includes apol, a tool with similar capabilities as the command-line tools, but with a graphical user interface. Policy developers commonly use text editors such as vi to write policy, and other command-line tools such as grep to search the policy. See Section 6 for a more detailed discussion of our survey on tools commonly used by policy developers. These tools can be useful, but due to the size and complexity of SELinux policies, they can be cumbersome for understanding the intricacies of even medium-sized policies. This is because the tools are primarily text-based, which places a higher load on cognitive processes and working memory than well-designed visualizations and graphical user interfaces.

Abstractly, we can consider SELinux security policies to be policies (policy types) with links (policy permissions) expressing relationships between them. This is a common abstraction and can be seen in several research projects (e.g., [8, 16, 15, 11]).

NodeXL\(^3\) and Gephi\(^2\) are common tools for analyzing and exploring graph datasets. These tools can be very effective, but they are not designed to show the differences between two graphs, nor do their design take into account the extra semantics found in SELinux security policies, such as subject types having a set of permissions on objects and classes.

SEGrapher\(^8\) is a tool designed specifically for visually analyzing SELinux policies. SEGrapher visualizes the policy as a node-link diagram, where nodes are types and links are policy rules connecting types. Because policies can be quite large and therefore challenging to analyze and visualize, SEGrapher employs a clustering algorithm to reduce the complexity of the visualization. The clustering algorithm clusters types based on a set of relations, and then allows users to see the clusters of object types that are accessed by user-selected subject types. This produces a smaller, simplified graph than showing all the types simultaneously, at the expense of not allowing users to see the entire policy at once.

Pan and Xu\(^11\) proposed using treemaps and semantic substrates for visually analyzing multi-domain access control and cross-domain information exchange, a related but different usecase from MAC. Xu et al\(^16, 15\) also used semantic substrates, but in the specific context of SELinux security policies. They use semantic substrates to show users, roles, domains (subjects), and types (objects). Xu et al supplemented semantic substrates with adjacency matrices to show paths between types. However, it is unclear whether these visualizations would scale to show large portions of a policy, and they were not designed to show the differences between two policies. To help analysts identify potential policy violations, they also implemented a visual query system intended to make it easier for typical system administrators to execute policy queries.

Chen et al\(^3\) extract attack graphs from SELinux and AppArmor policies, identify the minimal attack paths, and compare the Quality of Protection between different policies using the minimal attack paths. This approach is useful for identifying weaknesses in a system, but it is not designed to show all the differences between two versions of a policy.

Clemente et al\(^4\) propose a method to analyze logs of attacks on SELinux systems, and then visualize the information flows during attacks. The information flows are visualized as node-link diagrams, and edge color indicates the frequency of events occurring during attacks. These information flows can be useful, but their system requires access to system log files, which may not be available to policy designers and analysts. Their work was also not designed to show differences between policies.

Wang et al propose a semi-supervised machine learning approach to analyze SEAndroid audit logs for the purpose of refining an existing policy. This is a worthwhile approach, but it does not help policy developers understand existing rules in a policy, and it is not designed to help analysts understand the differences between two policies.

Expandable Grids\(^13\) is an interaction technique for visualizing and authoring security policies. It is designed to show the combined set of permissions that a subject will have on an object, and they allow users to expand the hierarchy to show more detail (e.g., to expand a directory to show permissions on individual files).

Outside of the context of security policies, TreeVersity\(^5\) was designed to show differences between pairs of trees. In particular, TreeVersity shows differences in tree structure and numerical node attributes. SELinux security policies do not have a strong sense of hierarchy, and policy graph nodes, such as types or classes, do not typically have numerical attributes that would change between policies. TreeVersity also is not designed to scale to the thousands of nodes often present in SELinux security policies.

TreeJuxtaposer\(^10\) is designed to be able to compare structural changes between trees with hundreds of thousands of nodes.

Visualization of temporal network evolution, also known as dynamic networks, is used to compare changes in the structure or attributes of a graph over time. There are many powerful techniques for dynamic network visualization (see Bach et al\(^1\) for a survey of several such techniques), but in general they are not designed to adequately capture changes in the unique semantics found in SELinux policy graphs.

4 V3SPA DESIGN

V3SPA is designed to analyze SELinux and SEAndroid binary policies. V3SPA can also import and analyze SELinux policy source for Linux systems that is in the reference policy format by making use of the Lobster\(^6\) domain specific language, but V3SPA has similar features for analyzing policy binaries versus reference policy source, so this paper focuses on the support for SELinux and SEAndroid binary policies.

V3SPA is primarily concerned with the type enforcement elements of SELinux security policies. In particular, V3SPA analyzes and visualizes the allow rules from security policies.

4.1 System Architecture

V3SPA is architected with a web browser-based frontend and a web server-based backend. The backend is written in Python using the Tornado web server framework and the MongoDB database. The frontend and the backend communicate via REST API calls and websockets. When users load the V3SPA frontend in their browser, they can import an SELinux policy by uploading it in a zip archive. The zip archive can contain the policy binary, the policy source, or both. The frontend sends the policy to the backend through websockets, where the backend uncompresses the zip archive to disk, uses search from SETools version 4 to extract all allow rules from the policy binary, and then stores the text of the allow rules in the MongoDB database.

When the frontend issues a call to the backend to retrieve the policy explorer graph or the policy differ graph, the backend calculates the graph in two steps. In the first step, the backend parses the allow rule text to create a JSON representation of the allow rules. The parser extracts the sets of subjects, objects, classes, and permissions from each allow rule, computes the Cartesian product of the sets from each allow rule, and then produces one JSON object from each element of the Cartesian product. For example, in the allow rule **allow user_t bin_t : file { read write};**, the parser extracts one subject, one object, one class, and two

\(^2\)https://github.com/TresysTechnology/setools

\(^3\)https://github.com/TresysTechnology/setools
permissions. The Cartesian product of the subject, object, class, and permissions sets is a set of two elements, and the parser creates two JSON objects (shown in Listing 1), one for each element. After parsing all the allow rules, this produces an array of JSON objects that are then cached in the database to avoid re-parsing the allow rules.

```
[{
    "subject": "user_t",
    "object": "bin_t",
    "class": "file",
    "perm": "read",
    "rule": "allow user_t bin_t : file {read \n    \nwrite}";
}, {
    "subject": "user_t",
    "object": "bin_t",
    "class": "file",
    "perm": "write",
    "rule": "allow user_t bin_t : file {read \n    \nwrite}";
}
]
```

Listing 1: The JSON representation for the allow rule

```
allow user_t bin_t : file {read write};
```

In the second step, the backend calculates the policy explorer graph, or the policy differ graph, depending on the request. If the graph is already stored in the database, the backend will send the previously calculated graph instead. If it is not already stored, it calculates the graph and caches it in the database before sending it to the frontend.

To calculate the policy explorer graph, the backend iterates over each JSON object created by the parser. The backend creates one node for each subject, and one node for each pair of objects and classes that both appear in the same rule (i.e., for the rule `allow user_t bin_t : file {read write};`, the backend would create one node called `bin_t.file`). The backend creates an edge linking a subject node and an object-class pair node if the subject has at least one permission on the object-class pair.

To calculate the policy differ graph for a single policy, the backend iterates over each JSON object and creates one node each for the subjects, objects, classes, and permissions in the policy. Edges connect subject nodes to permission nodes, permission nodes to object nodes and class nodes, and object nodes to class nodes if these policy components occur in the same allow rule. In the running example `allow user_t bin_t : file {read write};`, the backend would create five nodes: one subject node `user_t`, one object node `bin_t`, one class node `file`, and two permission nodes `read` and `write`. The backend would also create edges from the `user_t` node to the `read` and `write` nodes, from the `read` and `write` nodes to the `bin_t` and `file` nodes, and from the `bin_t` node to the `file` node. This graph structure mimics the way one might read the allow rule: The `user_t` domain has `read` and `write` permission objects of the `bin_t` type with the `file` class. Note that the differences between two policy graphs are not computed by the backend; they are dynamically on the frontend when two policies are loaded.

Graphs on the backend are stored in the JSONH\(^3\) format, which converts an array of JSON objects into a CSV-like array. For example, consider the following array of edges:

```
[
    {
        "source": 0,
        "target": 1
    },
    {
        "source": 2,
        "target": 0
    }
]
```

The JSONH representation would be

```
[2, "source", "target", 0, 1, 2, 0]
```

This is much less verbose and consequently significantly fewer bytes in size. When the frontend receives the JSONH representation of a graph, it first converts it back to JSON before further processing the graph.

4.2 System Performance Optimizations

The V3SPA system architecture includes several designs to improve performance. It is quite common to have graph datasets with tens of thousands of nodes and edges, but many readily available graph analysis tools have difficulty achieving good performance on graphs of this size. In an attempt to aid other system designers in building performant graph analysis systems, both for cybersecurity-related and unrelated domains, I report on several design decisions that enable V3SPA performance to scale up to visualizing entire SELinux policy datasets with large numbers of nodes and edges.

**Caching.** In order to avoid rerunning expensive algorithms, V3SPA makes extensive use of caching on the backend. Specifically, V3SPA caches the parsed JSON version of the allow rules, and the policy explorer and policy differ graphs calculated from the JSON allow rules. These operations can take tens of seconds, so caching the results in the MongoDB database decreases the load time when the policy is loaded again in the future.

**Data compression.** JSON is a very verbose language, which can cause datasets to become very large in size and cause long delays in transferring the data from the backend to the frontend. Most of V3SPA's data is stored as arrays of objects with identical structure, meaning that they repeat the same key names over and over, which dramatically increases the size of the data. This data size can be reduced by using JSONH as described above, so that key names are not repeated unnecessarily.

In the particular case of graphs, such as those in V3SPA, the array of edges need to reference source and target nodes for each edge. Instead of referencing the nodes by name (e.g., `[source: 'nl-name', target: 'n2-name']`), the edges reference nodes by the nodes' indices in the array of node objects (e.g., `[source: 1, target: 2]` references the nodes at index 1 and index 2 in the array of nodes). This is a convenient method for referencing nodes, and therefore not uncommon, but in this particular case it also greatly reduces the size of the array of edge objects, especially when combined with the JSONH format.

Finally, some node and edge properties are needed for the visualization, but they are repeated throughout all of the data for a given policy and can quickly and easily be inferred on the frontend. For example, the policy differ needs to know the name of the policy that the subjects, objects, classes, and permissions come from. However, the frontend already knows the name of the policy when it requests the policy differ graph from the backend, so it can quickly add those attributes to the data on the frontend instead of relying on the backend to do so.

---

\(^3\)https://github.com/WebReflection/JSONH
When taken all together, these techniques provide a massive reduction in size of data to be sent from the backend to the frontend, in many cases reducing the size of the data from tens of megabytes down to a few hundred kilobytes. This drastically improved the load time for each policy dataset.

**Canvas rendering.** Many web-based visualizations are rendered using SVG, where roughly each visual element corresponds to one element in the browser’s document object model (DOM). There are many advantages to this approach, but browsers often experience considerable rendering performance loss when there are more than a few thousand DOM elements. Canvas is a popular alternative to SVG, where the visualization is rendered as a bitmap, which reduces the rendering burden on the browser. The policy explorer view, which displays tens of thousands of nodes and edges, takes advantage of this to avoid suffering a frustrating rendering performance loss. However, the policy differ renders the differences between two policies using SVG. This could cause some performance issues when visualizing all the components of two policies simultaneously, but in practice this has not been an issue. This is because when comparing two policies, users often focus on the differences between the policies, which are typically a small percent of the components in the overall policy. As a result, the number of visible SVG elements at any given time is typically small and not a performance bottleneck.

**Data representation.** The scheme for mapping policy components to graph nodes also has an impact on performance. The policy explorer represents pairs of objects and classes as an individual node, as opposed to having separate nodes for each. This adds roughly 14,000 nodes in the graph compared to having separate nodes for each object and class, but it also reduces the number of edges in the policy explorer graph by about 33%, or about 40,000 edges. This is useful for performance reasons, but it also reduces the density of the graph, making it more readable.

### 4.3 Policy Explorer

On the frontend, when users select the policy explorer view, the frontend retrieves the policy explorer graph from the backend and renders the graph using the Sigma\(^4\) JavaScript library’s canvas renderer. As described in Section 4.1, the policy explorer shows a graph of subject nodes (dark blue) and object-class pair nodes (orange), where object-class pairs are created if the object and class both occur in the same allow rule. Edges connect subject and object-class nodes if the subject has one or more permissions on the object-class pair. The frontend uses a modified force-directed layout algorithm that is less accurate than Barnes-Hut force directed layout algorithms \([12]\), but computes layouts faster. Figure 1 shows a screenshot of the policy explorer.

The policy explorer includes several controls for filtering the policy. These filters are broken into several tabs.

The **Statistics** tab has several range sliders to filter the nodes in the graph. Users can filter by node degree, node authority score \([7]\), and node hub score \([7]\). Nodes with non-zero authority scores are object-class nodes, because they are the only nodes with incoming edges. Nodes with high authority scores are object-class nodes that have permissions on them by subject nodes that also have permissions on many other object-class nodes. Similarly, non-zero hub scores are associated with subjects, and nodes with high hub scores are subjects that have permissions on many object-class nodes. Therefore, authority and hub scores can be useful in identifying subjects and objects that may be over used in the policy.

The **Access vector** tab lists all of the subjects, objects, classes, and permissions found in the allow rules in the policy. Users can select specific components of access vector rules they wish to see or hide. For example, hiding specific subject nodes, or showing edges that represent specific permissions.

The **Denials** tab allows users to paste AVC (Access Vector Cache) denials and filter the policy to show only access vector components related to the AVC denial. By doing this, users can see allow rules related to the denial, and see how it would affect the policy by allowing the denied functionality, and see if there is a way to get around the denial.

The **Always visible** tab displays a list of nodes that are always visible, regardless of other filter settings. Users can type the name of a node and see a suggestion list of all nodes that contain that substring, and then select a node to add to the always visible list. Users can remove a node from the list by clicking the “x” delete icon next to a node name, or by clicking the “Clear all” button. Users can then click the button to add those alternate nodes to the always visible list. Users can also add the selected node and/or its neighbors to the always visible list.

### 4.4 Policy Differ

When users select the policy differ view on the frontend, they must also select which two policies they wish to see as a diff. The frontend then retrieves the policy differ graphs for both policies, and the frontend then merges the two graphs by computing which nodes and edges are in only the first policy, which are only in the second policy, and which are in both policies.

The policy differ renders the merged graph of the two policies using SVG (Scalable Vector Graphics), where the nodes are divided into four groups: The subjects, the permissions, the objects, and the classes. These groups are arranged from left to right in that order. This ordering mimics how someone might read an allow rule: This subject has this permission on this object type with this object class.

The policy differ borrows visual language from other diff tools, where a node with a left arrow indicates a node only in the first policy, a node with a right arrow indicates a node only in the second policy, and a node with a solid fill color represents a node found in both policies. This indicates to users whether a subject, permission, object, or class has been added or removed in a new version of the policy, or whether it was present in both policies. The nodes are also sorted so that nodes only in the first policy are displayed at the top, nodes only in the second policy are displayed at the bottom, and nodes in common between the two policies are in between.

When users hover the mouse cursor over a node, the policy differ highlights all other nodes representing policy components that co-occur in allow rules along with the hovered node. For example, if a user hovers over a subject node, then the policy differ highlights all of the object and class nodes that the subject has permissions on, as well as the corresponding the permission nodes. Edges connect the highlighted nodes to make the relationships more clear.

Note that this visual language cannot capture all differences between two policies. For example, if we have the following two rules in version 1 of a policy:

```plaintext
allow user_t bin_t : file read;
allow user_t bin_t : dir write;
```

And in version 2 of the policy those two rules change to become

```plaintext
allow user_t bin_t : file write;
allow user_t bin_t : dir read;
```

Note that this is not immediately visible with the current policy differ system.
There are no new nodes, but the relationships between the nodes changed. In particular, the write permission is now associated with the file class, and the read permission is now associated with the dir class. To see these differences in relationships, users can select to see only edges representing relationships only in the first policy, only in the second both, or edges that are in both policies.

Users can click a node to make the highlights and edges always visible, allowing users to pan and zoom the visualization, or hover over another node to see its associated nodes and edges, while still seeing the clicked node and its connections. Clicking a node also retrieves all of the allow rules associated with that node and displays them in the Details tab. Users can then see the rules, along with the name of the policy that contains each rule.

Users can also right click a node to get a context menu with several menu options to hide or show all connected nodes of each group (e.g. hide or show all permissions connected to a subject).

5 Example Analyses

To better illustrate how to use the policy explorer and the policy differ to examine SELinux security policies, I present the following example analyses.

5.1 Policy Explorer

To demonstrate the policy explorer view, I load the binary policy from the Nexus 4 device (occam 5.1.1 build ID LMY48T).

At a high level, we see that there are a few dyads, i.e. a single subject node with permissions on a single object-class node. There also appear to be many object-class nodes that have permissions from only a few subject nodes. (See Figure 3a.) I quickly confirm this by adjusting the connections slider to hide nodes that have number of edges (connections higher than 1,100), which causes two subject nodes to disappear along with their edges. Many object-class nodes now become disconnected (see Figure 3b). Similarly, by hiding all nodes with fewer than 4 connections, most object-class nodes disappear. These two highly connected subject nodes are the unconfineddomain node (connected to 1265 object-class nodes) and the init node (connected to 1142 object-class nodes).

If we tried to run this policy on our device and we got the following AVC denial, we could paste it into the denials tab and see all of the related policy components.

```
avc: denied { write } for pid=1003
name="kgsl-3d0" dev="tmpfs"
scontext=u:r:mediaserver:s0
tcontext=u:object_r:device:s0
tclass=chr_file permissive=1
```

Doing so, we see the mediaserver subject node, and several object-class nodes that have either the device object or the chr_file class (see Figure 3b). The mediaserver subject node is connected to the video_device.chr_file, rpsmg_device.chr_file, camera_device.chr_file, tee_device.chr_file, gpu_device.chr_file, and audio_device.chr_file nodes via the write permission.

The user could choose to check if the mediaserver has permissions on a particular object-class pair, such as the sdcard_type.file node. To do so, users can open the “Always visible” tab, and begin typing “sdcard_type” in the input box. An autocomplete list of suggested node names appears, and the user can select “sdcard_type.file” from the list. This node is then added to the list of nodes that are always visible, and the node appears in the visualization. It is connected to the mediaserver subject node, and the user can click on the sdcard_type.file node to see all of its allow rules in the “Details” tab (see Figure 3c). There are six allow rules, and we see that one of them...
Figure 4: (a) A context menu gives users options for filtering connected nodes. (b) After showing objects connected to the \texttt{redis\_log\_t} subject, we see it has the \texttt{associate} permission on the \texttt{tmp\_t} and \texttt{tmpfs\_t} objects.

Figure 5: Showing the connections between policy components that were removed after version 20130424.

gives \texttt{mediaserver} write permission on the \texttt{sdcard\_type} object and \texttt{file class}.

If the user decides to add rules to address the AVC denial, the user could modify the policy, build it, re-import it, and then use the policy differ to verify that the changes only affect the desired portions of the policy.

5.2 Policy Differ

In this example, I load two consecutive versions of the Tresys reference policy. First I load the 20130424 release, and then I load the 20140311 release to compute a diff between them.

When the policy differ initially loads, it defaults to showing only policy components that have been added or removed from one policy to the next. We see that two subject nodes with left arrows, indicating that these are subjects from the 20130424 policy that have been removed in the 20140311 policy. Meanwhile, 46 subjects have been added to the 20140311 policy. Similarly, we see 6 objects that were removed in the 20140311 version, and 59 objects that were added. There are no added or removed permissions or classes.

In the “Nodes” tab, we can select to see all of the permissions and classes. By doing so, we can hover over a subject and object node and see the other policy components associated with it. For example, we can hover over the \texttt{redis\_log\_t} subject and see that it has \texttt{associate} permission on itself and \texttt{filesystem} class (see Figure 2). This was policy functionality that was added in the 20140311 version.

One usecase of the policy differ is to verify that policy changes do not accidentally introduce unexpected behavior. For example, we might wonder if the \texttt{redis\_log\_t} has any permissions on other objects. To examine this, we can right click on the \texttt{redis\_log\_t} subject and click the “Show connected objects” menu item (see Figure 4a). Two objects are added to the objects group: the \texttt{tmp\_t} and \texttt{tmpfs\_t} objects (see Figure 4b). Both of these objects are in both policies, as indicated by the fact that they are rendered as a solid-colored node. This indicates the \texttt{redis\_log\_t} has the behavior that we would expect, and there is nothing out of the ordinary.

To see changes in relationships between subjects and objects that exist in both policies, we can show only the nodes in common to both policies, make the links always visible, and show only the links in the 20130424 policy (see Figure 5). This reveals that there are a small number of relationships that have been removed in the 20140311 version.

6 Policy Developer Survey

To compare V3SPA to the way policy analysts and developers currently work with SELinux policies, I conducted a survey of five people with SELinux policy analysis or development experience. This is not a large survey and thus the responses may not generalize well across all policy developers and analysts, but this information can help us understand some of the current problems faced in the real world.

The survey respondents had a range of experience working with SELinux or SEAndroid policies, from 1–2 years of experience up
to 5+ years. Three of the respondents have worked primarily with SEAndroid, one mostly with SELinux, one about the same amount of experience with both. Three of the respondents mostly have experience modifying policy, and the other two have about equal experience modifying and analyzing policies.

The survey respondents mostly use text editors to edit policy (nano, vim, and gedit). One respondent reported using audit2allow to generate new rules from audit logs, and two respondents reported using sepolicy-inject to add rules to a binary policy.

For analyzing policies, four respondents reported using grep, making it by far the most commonly mentioned tool. Respondents also use dmesg to monitor AVC denials, seinfo and sesearch (reported by two respondents each), and apol (reported by one respondent).

When asked what was the most time consuming task of analyzing a policy, two respondents indicated that searching across the many source files in a policy was the most time consuming. The other most time consuming tasks were understanding the structure of a policy, understanding policy variation across devices, understanding the behavior of the policy, and understanding the policy build tools and macros (each answer was given by one respondent).

From these responses, there is not much variety of tools used to edit policies. Mostly respondents use text editors. This makes sense, but indicates that policy developers might benefit from tools that provide code completion, refactoring support, syntax highlighting, or other tools found in modern programming environments.

There was more variation in the tools used to analyze SELinux policies. Searching appears to be a very common task, as indicated by the number of respondents that use grep and the one respondent that reported using search. V3SPA has some support for searching in the policy explorer view, and both the policy explorer and the policy differ show users the full list of all subjects, objects, classes, and permissions, which can help users find a policy component if they are not sure of the name or where it occurs in the policy. These tools are primarily text based, which are not well suited to displaying large amounts of information that would be helpful when working with a large, complex dataset such as an SELinux security policy. This might explain why so many respondents reported that the most time consuming tasks when analyzing a policy have to do with searching and understanding the big picture of a policy. V3SPA was designed to address these issues by showing an entire policy at once, helping users identify a portion of the policy of interest, and narrowing their focus to identify specific details of the policy.

7 Conclusion
This paper presents V3SPA, a novel tool for exploring, analyzing, and differ SEAndroid security policies. V3SPA has two main visualizations: A policy explorer, and a policy differ. The policy explorer is designed to help people who work with SELinux policies understand their policy better, a task that our survey respondents say is time consuming with the current ecosystem of tools. The policy differ is a novel visualization that supports differential policy analysis by showing a visual diff of two versions of a policy.

In addition to the visualizations, this paper makes several other contributions. I discuss several system architecture design decisions that enable V3SPA to scale up and visualize SELinux policy graphs with tens of thousands of nodes and edges. I also present a survey of five SELinux and SEAndroid policy developers and analysts. The results of the survey indicate a need for powerful tools like V3SPA to help analysts understand the big picture of the large, complex security policies found in SELinux and SEAndroid.

One avenue for future work is to design a better grid layout for the policy differ. Currently, the groups of nodes get crowded when there are many nodes visible. The policy differ would benefit from an elastic layout that dynamically adjusted the size of each group to accommodate the number of nodes currently in view.

Acknowledgements
I would like to thank the V3SPA team for their support and technical contributions: Jeff Karrels, Kristina Creque, Daniel Su, Christopher Wacek, and Matthew Oertle. I would also like to thank Joe Tanen for valuable feedback and discussion, and the survey respondents for their time and insight.

References