Continuous Model-Based Verification of the Linux Kernel

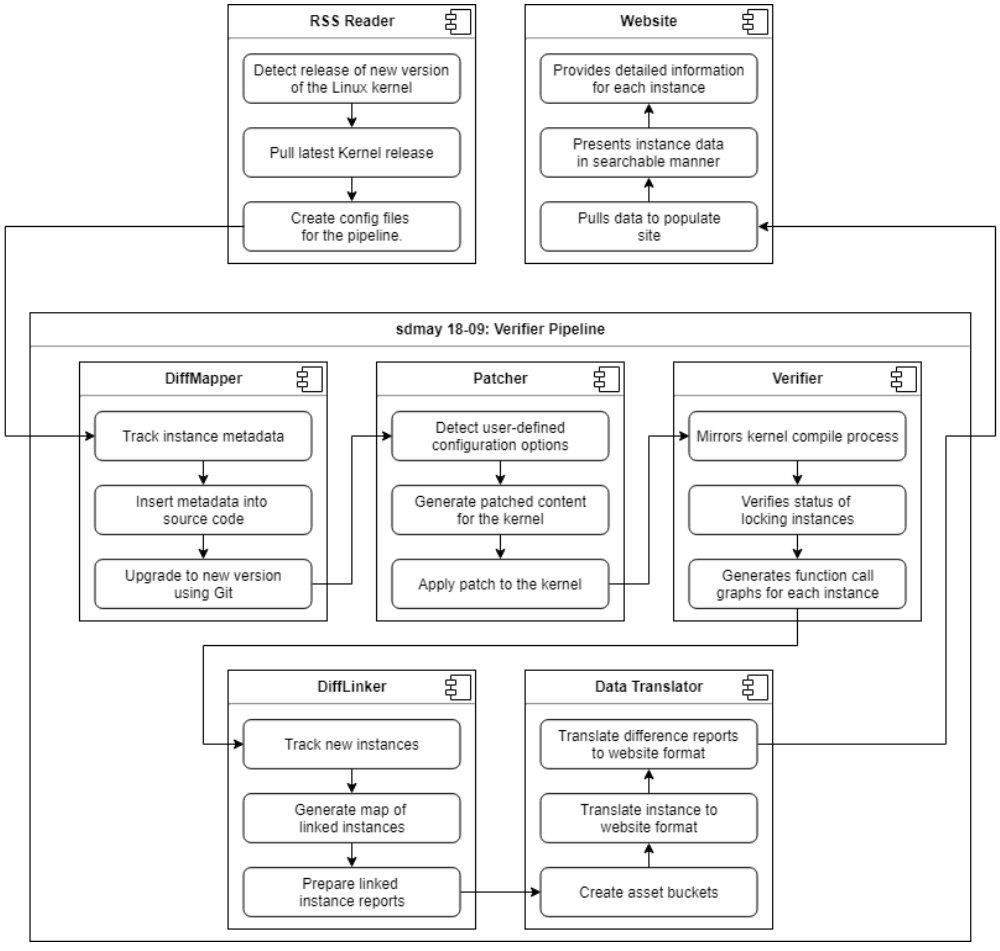
Final Report

Team #9
Client/Adviser: Dr. Kothari
Srinivas Dhanwada - Team Leader
Collin McIntyre - Scribe, Tool Integration Leader
Matthew Wall - Web Leader
Ben Weno - Automation Leader
sdmay18-09@iastate.edu
http://sdmay18-09.sd.ece.iastate.edu/

Created: 4/23/2018 | Version 1

Project Design

We approached this problem with configurability in mind. We wanted each part of our pipeline to be as configurable as possible while still being easy to use. We split the pipeline into seven components, which are listed below in the technical details. The verifier, or L-SAP itself, was provided by our client, and we implemented the other six components. Each of the components comes with several configuration options for manipulating/displaying data. We aimed very much for a pipeline approach, wanting each of our components to execute sequentially rather than have multiple components execute simultaneously. This makes it easier to determine which part of the pipeline is malfunctioning if something isn't working correctly, and also makes it easier to implement a script or other program that executes the entire pipeline without requiring manual input. A description of each component is found below.



RSS Reader

The RSS reader serves as the beginning of our pipeline. It polls the kernel release's rss feed, looking for when a new release becomes available. From there, it automatically pulls the new kernel, and generates a config file to be used by the rest of the modules.

DiffMapper

The DiffMapper project aims map metadata between two version of the Linux Kernel. L-SAP completes a run and provides results for a specific version of the Linux Kernel. However, L-SAP does not support generating a report of the differences between two versions of the kernel. This is due to number of locking instances in the kernel and the potential changes that could occur from version to version of the kernel. Between two versions, we can't guarantee that the same number of locking instances will appear in each file. Further, we can't guarantee the relative order of locking instances in a file. Because of this, we need some way to map result data that will satisfy the following constraints:

1. If a locking instance changes locations in a newer source file, there is still a link back to the old location in previous versions.
2. If a locking instance has its variable name changed in a newer source file, there is still a link back to the old location in previous versions

Because of these constraints, we can't simply look for the same named instance in a file or look at the same location of a file to determine whether or not the same instance exists between versions. A deeper solution is needed to solve this problem.

DiffMapper leverages git to handle tracking instances as they move between versions. Because the kernel is developed using git, we have a very robust record of how source files change over time. git tracks not only movements of code between files, and inside files, but it also tracks edits to name changes too! Because of this, as well as git's efficient implementation, we can insert metadata within the source code and have git do the heavy lifting for us.

The basic steps of DiffMapper are as follows:

1. Checkout the old version of the kernel
2. Use L-SAP results of the old version to find the correct locations of all instances in the kernel
3. Insert the L-SAP results as comments in the source code (referred as *metadata*)
4. Use git to upgrade to the new version of the kernel - This allows the metadata to follow along with the code changes

   At this point, the instances from a previous version will be "mapped" and the kernel can be passed to L-SAP.

Patcher

The Patcher generates a patch for the Linux kernel that allows L-SAP to run as efficiently as possible. The Patcher creates two new header files for the kernel: `lsap_mutex_lock.h` and `lsap_spin_lock.h`. These header files define empty body functions for each type of lock and macro wrappers that redirect existing locking function calls to the new empty functions. The Patcher also reads through files in the kernel that define or implement the existing (unpatched) locking functions and removes them. The Patcher will then output the generated and modified files to a directory the user chooses.

The Patcher does not add new functionality to our client's existing product. The Patcher automates a long and tedious manual process that's required for the Verifier to run properly.

Verifier

The Verifier, also known as L-SAP, is a program that was developed by our client. It ensures that each resource lock is accompanied by an unlock through every path the code can take. It also generates human readable graphs for each locking instance, allowing users to determine if a lock is paired or unpaired when L-SAP isn't able to determine this on its own.

DiffLinker

The DiffLinker is built as a companion piece to the DiffMapper and is meant to wrap around L-SAP to handle tracking instances across versions. L-SAP does not support tracking instances between versions, however DiffMapper allows instance metadata to be embedded into the kernel source code. Using a specific git merge strategy, the metadata is able to traverse the source code along with the instances. Therefore, the results of L-SAP can be analyzed to determine the source code location of instance metadata.

If metadata exists, DiffLinker captures this and produces a "link" between versions. If the metadata does not exist, DiffLinker attempts to perform linking by analyzing instances for a certain file and comparing those instances. This type of linking is more expensive to perform, but does not get performed often enough to have a degradation in performance.

The DiffLinker generates several reports that show the "linked" instances between versions as well as a subset of "interesting" cases for researchers to focus on when analyzing the instance data. These instances are not guaranteed to be reveal new insights in to the kernel, but are an attempt by the DiffLinker to point researchers to instances that might provide insights.

The basic steps to DiffLinker are as follows:

1. Generate New Instance Map
2. Analyze the Kernel Source Code for Metadata
3. Generate Reports for Linked Instances and “Interesting” Cases

Data Translator

The data translator takes the output from the verifier and translates it into a format that can be uploaded and used by the website. This consists of 3 components:

- A JSON file to be uploaded to the database, consisting of instance data, and links to images
- A JSON file also uploaded to the database containing all the information for links between the new version and the previous
- An asset bucket containing the proper directory structure containing images that matches the links in the first json file.

Website

Overview & Purpose

The website’s core purpose is to be an endpoint for the researchers running L–SAP and for people who are curious about the security flaws in the linux kernel. It’s functionality is to provide a way to sift through large amounts of data very quickly. They’ll also be able to view the data in an intuitive, user-friendly way that makes sense to both: the researchers and the common user.

Design Details

Originally, the website the KCSL (Knowledge Centric Software Laboratory) had was a bunch of static links with static file includes. Our goal was to turn this into an inclusive, version controlled, and scalable way to view the output from our verifier. We built the basis of the site on an angular 5 framework. This allowed us to render components separately, which gives us the advantage of faster loading speeds when we have large amounts of data on individual pages. We also chose to use typescript for the main programming language as opposed to vanilla javascript. The support for angular & typescript is pretty substantial which is also very helpful. Within the project, we used many dependencies such as: Bootstrap 4, AngularFire2, and many more.

Implementation Details

RSS Reader

When the RSS reader is first run, it will try to pull the newest kernel, if it hasn't already been pulled to the selected location. Next it will enter into a loop that will poll the kernel rss feed periodically based on the parameter the user passed in. Each time it checks the rss feed, it will parse the xml file, looking at all the version entries and compare it to the current pulled version. If a new version is detected, it will pull it down, then generate a diff config to be used by later modules in the pipeline.

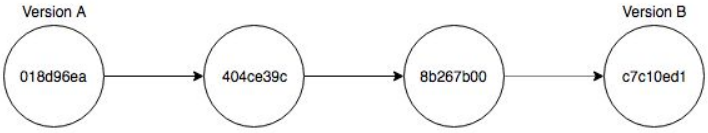
DiffMapper

The following sections provide a detailed overview of each step. The following data is stored in a Config object and is required by DiffMapper to complete successfully.

- Kernel Location - The location of the kernel we want to manipulate is required by DiffMapper. This kernel is required to be a local clone of the Linux Kernel Repository
- L-SAP Result Location - DiffMapper uses the results from L-SAP to generate and insert metadata.
- DiffMapper Workspace - In order to support running DiffMapper on various directories from a single location, DiffMapper a location where it can store the intermediate tracked instances. See below for more information on this data.
- Old Version Tag - Since DiffMapper uses the Linux Kernel's git history, the tag of the old version needs to be known in order to correctly check it out
- New Version Tag - Since DiffMapper uses the Linux Kernel's git history, the tag of the new version needs to be known in order to correctly upgrade from the old version to the new version
- Types of Locks - This option allows a consumer to specify what type of locks they want to map. This makes DiffMapper run faster, although it is recommended to add all types of locks to the mapping.

Checkout the old version of the kernel

DiffMapper invokes a git command to checkout the old version (i.e. git checkout v3.17-rc1). A clean is then performed to make sure the checkout is in the correct state. Finally, a new branch is made from this tag, so changes can be made without interfering with the original linux kernel git history.



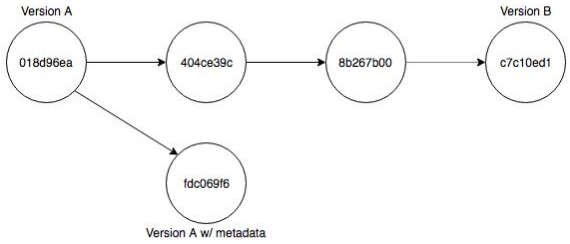
Use L-SAP results of the old version to find the correct locations of all instances in the kernel

Once the kernel has been prepared with the old version, We track instances from the input instance map and store them in memory to be inserted in a batch based on the file name. This map is necessary from two perspectives:

1. We don't know the order in which instances are tracked. It is easier for us for first create the metadata and store it based on the source file. Once we prepare all tracked instances, we now have a more cohesive grouping based on the filename.
2. Inserting comments requires file I/O operations that are much slower than storing data in memory. If we were to prepare a comment, then insert it directly in code, we would waste a lot of time opening a file and closing it multiple times just to insert one line. It is more efficient to open a file once, insert all metadata, then close it.

Insert the L-SAP results as comments in the source code

After we have metadata generated and stored based on filename, we must insert all metadata for a file in one batch. To do this, we must first sort the instances based on location. This is because when we enter metadata into the source file for one instances, we change all offsets for instances located later on in the file. If we sort instances based on their location, we can insert metadata and keep track of our induced offset so the metadata goes in the correct location. Metadata is inserted as a comment in the line above the location of the instance. This is to prevent our change from conflicting with any changes that could potentially occur.

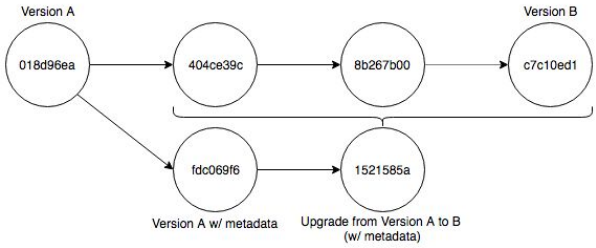


Use git to upgrade to the new version of the kernel

Once we have inserted our metadata, we add a new commit from our branch that tracks the source code changes. We then perform the following steps to "replay" all commits from the old version to the new version:

1. Checkout the new version
2. Perform a reset to remove the potential thousands of commits and only track the file differences between the two versions.
3. Perform a commit and create an "upgraded" branch, so we have only one commit between the old and new versions.
4. Rebase the "upgraded" branch onto our "metadata" branch so the git history starts from the metadata changes and then performs the necessary merges required to upgrade to the new version.

These steps will result in the code from the new version of the Linux Kernel, but with comments inserted nearby the previous locking instances.



Patcher

The Patcher operates in three phases: locking instance identification and header generation, header file inclusion, and implementation removal.

Locking instance identification and header generation

The Patcher begins by reading the contents of files listed in the configuration file under the "`MutexPathsToRead`" and "`SpinPathsToRead`" options. For each file, the Patcher identifies all functions and macros in the file, then checks the function and macro names against the criteria specified in the configuration file under the options "`MutexFunctionCriteria`", "`MutexMacroCriteria`", "`SpinFunctionCriteria`", and "`SpinMacroCriteria`". Of the functions and macros that meet the criteria, the Patcher generates two header files, one for mutex locks, the other for spin locks. These header files include empty implementations of functions that were identified and macro definitions of identified macros that redirect to the empty functions.

Header File Inclusion

Using the locking functions and macros identified in the previous step, the Patcher then reads the contents of the files specified in the configuration file under the options "MutexFilesToIncludeHeaderIn" and "SpinFilesToIncludeHeaderIn". The Patcher identifies the first function and macro in the file that's included in the patched header files, and inserts an include statement on the latest line possible before the earliest locking function or macro that will be removed in the next step.

Implementation Removal

In the Patcher's final step, it iterates through files in the kernel identified in the configuration file under the options "`MutexPathsToChange`" and "`SpinPathsToChange`". The Patcher identifies all functions and macros in these files that are included in the patched header files, and comments out their implementations and definitions.

When running the Patcher, the user can specify command line parameters that affect where the Patcher looks for data, puts data, and how it displays the data. The user can specify the location of the kernel to be patched, the location to store the patch (the user can specify the same location as the kernel to overwrite the existing files), and can enable various levels of debugging using the “debug” and “verbose” options

Verifier

The verifier, L-SAP, is a program that was developed by our client. While it is an important part of our pipeline and solution, we did not develop it and cannot provide implementation details. For more information on L-SAP, please visit the Iowa State University Knowledge Centric Software Lab website.

DiffLinker

The following sections provide a detailed overview of each step. The following data is stored in a Config object and is required by DiffMapper to complete successfully.

- Kernel Location - The location of the kernel that was used by L-SAP to perform a run. This kernel is required because it contains the metadata links from the DiffMapper.
- L-SAP Result Location - DiffLinker uses the results from L-SAP to read.
- DiffLinker Workspace - In order to support running DiffLinker on various directories from a single location, DiffLinker a location where it can store the intermediate tracked instances. See below for more information on this data.
- Old Version Tag - Since DiffLinker uses the Linux Kernel's git history, the tag of the old version needs to be known in order to correctly label linked instances
- New Version Tag - Since DiffLinker uses the Linux Kernel's git history, the tag of the new version needs to be known in order to correctly label linked instances
- Types of Locks - This option allows a consumer to specify what type of locks they want to map. This makes DiffLinker run faster, although it is recommended to add all types of locks to the linking.

Generate New Instance Map

Since the DiffMapper has been run previously, we have a map of all of the tracked instances in the old version of the Linux Kernel. Similarly, we need to generate an instance map for the new version of the Kernel. This is done in a similar way, except we look at the most recent results of L-SAP rather than the previous results.

Analyze the Kernel Source Code for Metadata

Once we have created our map of instances from the new version of the Kernel, we can use this map to reveal the source code locations of the instances. Since we have the kernel that includes the metadata, we can look through the source code for instance metadata from the previous version. If the metadata exists around the source code location for an instance, we know we have a link from the previous version to the current

version. If no instance metadata exists, there are a couple of reasons: 1) This instance is a new instance and did not exists in the previous version; 2) The instance metadata was not able to tracked to the new version. We determine the separation of these two instances by looking at the map of instances in the old version. We can compare the raw metadata of old instances in a single file to the new instances. Since the metadata tags are inserted for ~85% on average. This search is only done on single files sparingly, thus performance is not degraded.

Generate Reports for Linked Instances and “Interesting” Cases

Once we have analyzed the source code and linked instances. We generate a JSON formatted report that represents the new and old instances that have been linked. Instances that do not link back to old instances are formatted separately. After this report is generated, we perform another analysis on this set of data. As a starting heuristic, we look at the status of linked instances. If an instance exists that has different pairing statuses between versions, this case is marked as “interesting” and a separate report is generated to contain these cases. Such a report is a valuable tool to tell researchers an interesting subset of instances to focus on.

Data Translator

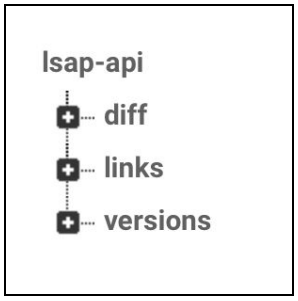
The data translator reads the results from the verifier and generates it’s output simultaneously. To do this, it first combines the spin and mutex folder, and iterates through each folder within them. For each folder, it uses some text parsing to pull the important information off of the name of the folder, and builds a json object containing this information. It also creates the asset directory structure and copies the images to the right places. Finally, the translator parses the diffs JSON file created by the DiffLiniker and translates it into a format that can be uploaded to the database.

Website

The site is really split up into two main parts (as with many websites): the front end portion, and the backend services.

For the front end, we’ve mentioned it briefly, but the framework we decided to use is angular with a typescript focus. This allows us to split up the major components of the website into, well, exactly what that means, *components*. This reduces code redundancy as well as helps optimize the overall speed of the website when loading such large data sets as the linux kernel provides. We chose to use typescript because it handles javascript objects a lot better than plain vanilla javascript does. This allows us to modularize the incoming data and verify it’s the object type we want rather than just throwing typeof calls everywhere to double check this.

The backend services are implemented through Alphabet's firebase realtime database. In order to communicate between the angular framework and firebase, we used a public API named angularfire2. This allows us to query data and return either a promise or subscription to the frontend which is receiving that data. But more about the data that we actually stored on the backend. The whole advantage of using a JSON-structured, NoSQL database is it really pushes us to keep the data sets in a flattened pattern. So instead of having many layers of data, as you would in a common SQL database, we have a few layers and but many top level structures. Here's a brief picture showing the upper level structures. Going further into the diff structure, we can see that the data is pushed to the database by a self-generated UUID per diff matching (pulled straight from the diff mapper data). This is very flattened and very fast to query and access. And all the rest of our upper-level structures have this implementation where we can very quickly pull data from these JSON formatted structures.



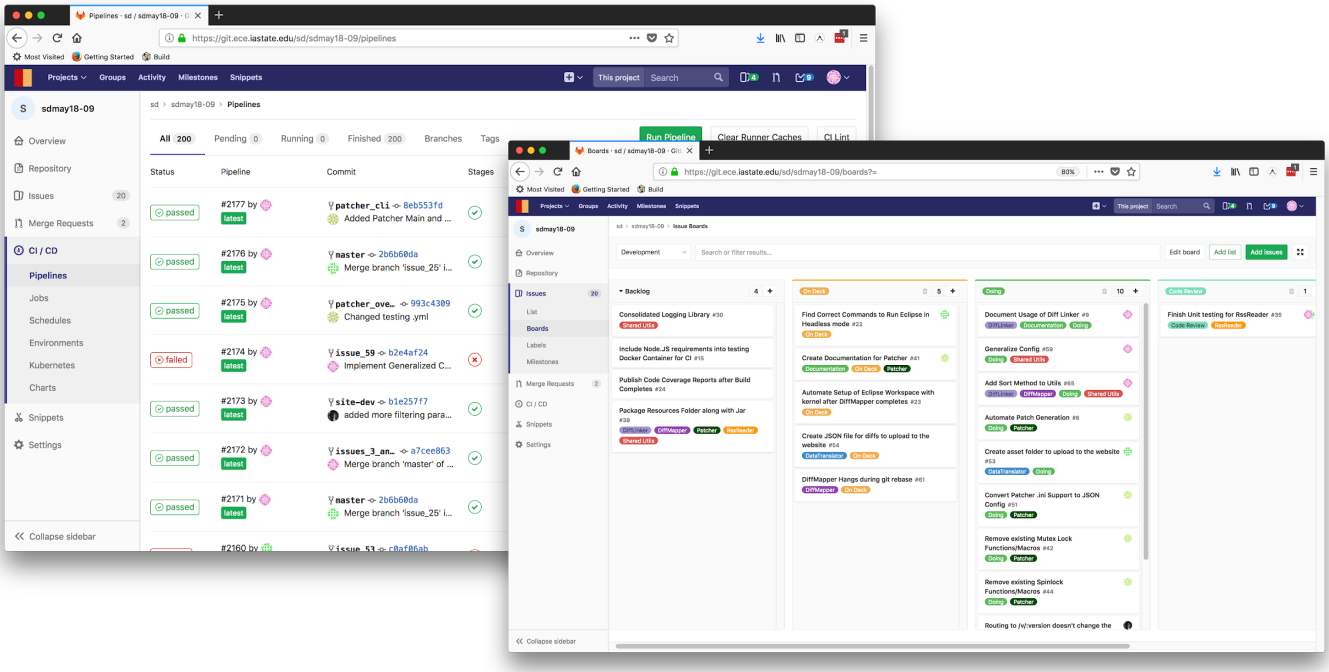
Also, our backend service stores our file data as well. Using Firebase's storage hosting we uploaded all the files in a structure that our data translator outputs. Then, again on our front end, we are able to load the images in a lightning fast way for thousands upon thousands of images.

Testing Processes

Our team used unit testing as our primary testing structure. This was effective due to the modular nature of our pipeline. Each project contains two directories for code: `src/main/java` and `src/test/java`. The source code for the module itself was located in the src/main/java directory for the project, and the unit tests were located in the src/test/java directory. We also made use of GitLab's built in Continuous Integration system. This allowed us to run the unit tests for every project on our remote repository whenever code was pushed to it.

To update the code on the master branch of our repository, we made use of other branches for completing individual issues. When the issue was completed and ready to be merged into the master branch, two of our necessary checks relied on testing. First, the unit tests for each project need to complete successfully on the developer's local machine. Second, the unit tests for each project need to complete successfully on the remote repository. This ensured that the projects functioned correctly regardless of the project workspace on any individual machine.

Another feature we included in our testing process was the use of a library called Jacoco. This was used for monitoring and improving the overall coverage of our unit tests. When the tests run, Jacoco monitors which lines of code are executed and helped us to develop tests that covered parts of the code we wouldn't think to test normally. Jacoco does this by generating an html document that shows the coverage for each class in a given project. It monitors both lines that were and were not executed as well as whether or not all possible branches were taken at branching points in the code. For example, for a given if statement, Jacoco will display if the code inside the if statement was always executed during tests, never executed during tests, or if the tests covered both possibilities.



Appendix 1: Operation Manual

RSS Reader

To use the RSS Reader, simply run the jar, the rest is automated. It supports the following command line parameters:

- `-Drun_once=<bool>` : If true, will only check the rss feed once
- `-Dgit_url=<url>` : Sets the git repository to try to pull from, defaults to https://github.com/torvalds/linux.git
- `-Dfeed_url=<url>` : Sets the location of the linux rss feed, defaults to https://www.kernel.org/feeds/kdist.xml
- `-Dworkspace=<dir>` : Sets the workspace directory to output the config file to, defaults to home directory
- `-Dwait_period=<int>` : Sets wait period in milliseconds between checks of the rss feed

Alternatively, these property values can be stored in a json formatted file and this file can be passed as as command line argument using `-c /path/to/config.json`.

DiffMapper

To use the DiffMapper, simply run the jar, the rest is automated. It supports the following command line parameters:

- `-Dold_tag=<tag>` : The git tag name of the starting version for the Linux Kernel
- `-Dnew_tag=<tag>` : The git tag name of the ending version of the Linux Kernel
- `-Dkernel_dir=<dir>` : The location of the local kernel git repository (defaults to `kernel/`)
- `-Ddiff_test_dir=<dir>` : The directory where instances maps are stored (defaults to `diffmap/`)
- `-Dresult_dir=<dir>` : The directory where results from a previous run of L-SAP are stored (defaults to `diffmap/prev_results/`)
- `-Dtypes.<idx>=<dir>` : An array encoded by specifying the name (`types`) and the index of the type of locks to map (defaults to `[mutex, spin]`). An example of this is `-Dtypes.0=mutex`

Alternatively, these property values can be stored in a json formatted file and this file can be passed as as command line argument using `-c /path/to/config.json`.

Patcher

Many configuration options are available in the Patcher's configuration file. The configuration file is a JSON file that allows the user to specify criteria that defines what identifies a function or macro as a locking function or macro, identify locations in the kernel that contain existing locking function/macro definitions and implementations, include custom functions/macros, and specify where in the existing

project the new header files should be inserted. See the Patcher documentation on our project repository for more information.

The Patcher is a command line application that also accepts various command line parameters. These allow the user to specify options such as the path to the kernel being patched, the path to the directory where the output should be stored, and enable various levels of debugging information. See the Patcher documentation on our project repository for more information.

DiffLinker

To use the DiffMapper, simply run the jar, the rest is automated. It supports the following command line parameters:

- `-Dold_tag=<tag>` : The git tag name of the starting version for the Linux Kernel
- `-Dnew_tag=<tag>` : The git tag name of the ending version of the Linux Kernel
- `-Dkernel_dir=<dir>` : The location of the local kernel git repository (defaults to `kernel/`)
- `-Ddiff_test_dir=<dir>` : The directory where instances maps are stored (defaults to `diffmap/`)
- `-Dresult_dir=<dir>` : The directory where results from the recent run of L-SAP are stored (defaults to `diffmap/curr_results/`)
- `-Dtypes.<idx>=<dir>` : An array encoded by specifying the name (`types`) and the index of the type of locks to link (defaults to `[mutex, spin]`). An example of this is `-Dtypes.0=mutex`

Alternatively, these property values can be stored in a json formatted file and this file can be passed as as command line argument using `-c /path/to/config.json`.

Data Translator

The data translator will create a json file and an asset bucket within the given workspace directory after running it. The results from L-SAP should be in <workspace>/results/kernel, and the output from data translator will be placed in <workspace>/dist. To specify workspace use the following command line argument:

- `-Dworkspace=<dir>`

Website

You can use our website like any other website. Visit the link here[1] for our demo, or here[2] for the final implementation which will contain the full deployment in the near future.

[1] https://lsap-api.firebaseapp.com
[2] https://lsap.knowledgecentricsoftwarelab.com

Appendix 2: Alternative initial versions of the design

DiffMapper and DiffLinker

The initial design behind the DiffMapper and DiffLinker were to analyze the graphs that are generated by L-SAP. Originally, it was thought that L-SAP generated a report that contained the graph data in an encoded format. This graph data could be used to determine the similar nodes and edges in the graph and the results from this would be able to link similar instances. This method would require less space and no interaction from git since we would only be analyzing the results from L-SAP. However, this posed several performance problems. The main issue was that instances could move from file to file. This type of movement meant that this algorithm would not be able to narrow down the search of a new instance to link if no instance was not found in the same file as the original instance that matched. At a worst case scenario, this would scale quadratically with the number of instances. Because of this, an alternative method using git was used.

Patcher

The initial design behind the Patcher hasn't changed from the concept phase, but there was a period when the Patcher was suffering from feature creep. This caused poor cohesion between the different classes in the Patcher, unnecessary dependencies, and unreliable output. The current version of the Patcher was created using the same methods, but with a better defined structure that eliminates unnecessary dependencies and provides more reliable output.

Website

The minimum viable product was accomplished for the website. Along with this, we had plans for more functionality behind which I'll discuss briefly.

Users: Being able to sign in or sign up for the website. From there, you could have a member role which would allow you to upvote or downvote a locking instance. This would alert the super-user of a potentially false positive or something that's maybe messed up.

Interactivity: The idea behind this was the provide the user with a way to click through and into functions much like Atlas (tool provided by EnSoft) allows you to. We would have implemented a Neo4j playground and centered it around each data point that L–SAP outputs to our linker. That way we could easily click into each data set and see the version it's mapped to in the previous/next version of the linux kernel.

Appendix 3: Other Considerations

Throughout this project we tackled a variety of problems, some familiar, and some completely new to us. First off, the gitlab runner we decided to set up for continuous integration was very finicky. All of us have desktop PCs that we have at home that are basically always on, so we thought we should utilize them for CI in someway or another. The problem was that the gitlab runner was a bit buggy when trying to set it up on anything other than a linux environment. Therefore, we had to create a virtual machine to boot up an instance of Alpine and pass the gitlab runner into the machine in order for it to run correctly.

We also used Maven in all of our modules, which was the first exposure to it for some of us. We learned how useful it was for importing external libraries, as well as sharing code between our modules. This enabled us to reuse common code such as the code for reading or writing the config file, or scanning through the output files from L-SAP, which we did in multiple places.

We also learned that the support for node libraries are heavily populated with useful APIs that we should use instead of reinventing the wheel. Things such as Bootstrap are useful for quick and easy styling effects without having to do much CSS (which we are all fairly bad at). There are many more which are way easy to implement rather than trying to code our own.

One thing that we ran into problems figuring out was dealing with parsing C code within the project. During development of the Patcher, we used regular expressions to parse the code, but we realized that regular expressions are not able to parse the code perfectly in every situation. There are just too many edge cases in the entirety of the kernel to safely cover them all nicely.