L-SAP

TOOL SUPPORT FOR CONTINUOUS MODEL BASED VERIFICATION OF THE LINUX KERNEL

SDMAY18-09

SRINIVAS DHANWADA | COLLIN MCINTYRE | BEN WENO | MATTHEW WALL

CLIENT/ADVISOR: DR. SURAJ KOTHARI | PAYAS AWADHUTKAR

INTRODUCTION

Our motivation behind this project stemmed from a couple of different problems. One being, the lack of automation in this desired process, and the other: the requirement for manual verification of these locking instances. So, we wanted to provide a way to solve both of these problems in a way that was efficient (meaning the pipeline won't tack on too much time additional to the time it takes to run L-SAP), and comprehensive (an end-user, or even a researcher, can view the results and understand them). Following along, you'll see the different component aspects of our pipeline and how each part communicates with other another.

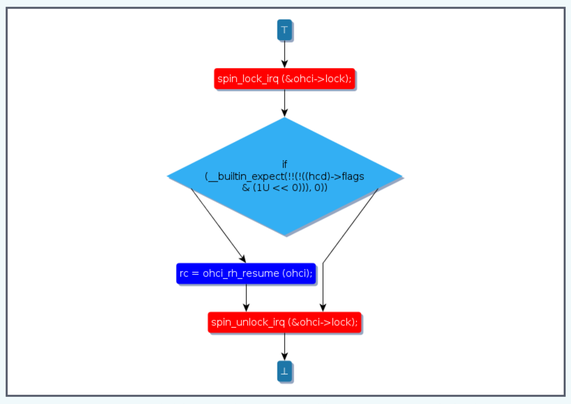

Figure 1: Human Readable Graph generated by L-SAP to verify lock status

DESIGN APPROACH

Figure 2: Full L-SAP Verifier Pipeline

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.

TECHNICAL DETAILS

rss reader

The RSS reader kicks off the pipeline by polling the kernel release RSS feed, and when a new release is posted, it automatically pulls the new release from github. Then it writes a config file for use with the rest of the modules, and starts the next step in the pipeline.

diff mapper

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 and inside files but also tracks in-depth changes, such as lock changes. Because of this, as well as git's relatively efficient implementation, we insert metadata within the source code and have git merge the new version on top of our comments. We then have a "mapped" kernel where instances from the previous version have "metadata" that is included in the new version.

patcher

The Patcher reads files in the existing kernel based on its configuration settings, detecting functions and macros in the kernel that are deemed "locking" functions and macros. The Patcher then generates two header files that redirect these functions and macros to a single empty locking function per lock type. After generating these files, the Patcher will also read files in the kernel that implement these functions and macros and comment out the existing implementations.

diff linker

DiffLinker uses the results from L-SAP to link instances between two versions of the kernel. Because we have a "mapped" kernel from the DiffMapper, DiffLinker uses the results from L-SAP to search the source code of the kernel at the location locking instances. If metadata is found at these locations, we track these links and generate a report of the findings. We also use some heuristics from the metadata to determine if a link is "interesting." These cases are stored in a separate report as well that allows researchers to investigate them further.

data transformer

The data transformer takes the output written by L-SAP and translates it into a format that we can upload and use on the website. The three things it creates are the json file containing data on all instances in the new version, the asset bucket containing all the images of the graphs in the proper folders, and the json file containing all the links between instances in this version, and instances in the previous version.

website

The website displays the outputted format from the data transformer. This is the last piece of the puzzle in the fully automated pipeline and allows the end-user to view the data efficiently. A big feature is being able to search through the data easily and in real-time. Here's a snippet of the homepage for version 4.13 of the kernel.

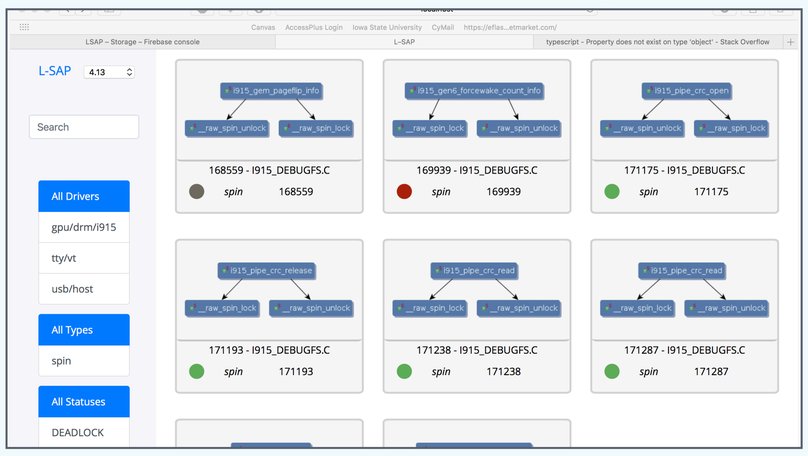

Figure 3: Website Instance List View

DESIGN REQUIREMENTS

Our motivation behind this project stemmed from the problem of both: the lack of automation in this desired process, and the requirement for manual verification of these locking instances. So, we wanted to provide a way to solve both of these problems in a way that was efficient (meaning the pipeline won't tack on too much time additional to the time it takes to run L-SAP), and comprehensive (an end-user, or even a researcher, can view the results and understand them). Following along, you'll see the different component aspects of our pipeline and how each part communicates with other another. and comprehensive (an end-user, or even a researcher, can view the results and understand them). Following along, you'll see the different component aspects of our pipeline and how each part communicates with other another.

ENGINEERING CONSTRAINTS

non-functional

Detect when a new release of the kernel is available, then automatically download and kick off the rest of the pipeline

Create and apply patch enabling use of L-SAP for each new release of the kernel

Map instances between each version of the kernel, making it easier to see what changed between releases

Create intuitive website allowing for a clear way to read and understand data from L-SAP

Format data from L-SAP and put into database to be accessed from website

functional

The website should be responsive, and provide a clear representation of the data outputted from L-SAP

Our tools will run on the same machine as L-SAP, and must be mindful of the memory and processing requirements of it

The pipeline should be started reasonably close to the release of a kernel, in order to keep results current

USERS

Researchers interested in running L-SAP

Researchers interested in viewing results

USES

Simplifies steps to run L-SAP

Provides methods to post results publicly

Automatically starts run of pipeline when new kernel is released

Link instances across versions to provide for easier interpretation of data

Provides a front end containing all the data from runs L-SAP

PROJECT MANAGEMENT & TESTING

Our project used a GitLab Repository to handle version control for the source code. We took advantage of GitLab by using it to handle issue tracking, feature requests, and code merges. Issues were placed on a Kanban Board and were assigned labels based on their progress and which modules were affected. This allowed us to see the progress of the project at a glance.

We also used on Discord to discuss our progress outside of meetings and ask/answer questions between team members. This allowed us to stay in near constant contact with each other and have a record of everything that was discussed. We also created a custom integration with gitlab to have it send us messages in a text channel based on certain events (issue creation, merge requests, pipeline failures, etc.)

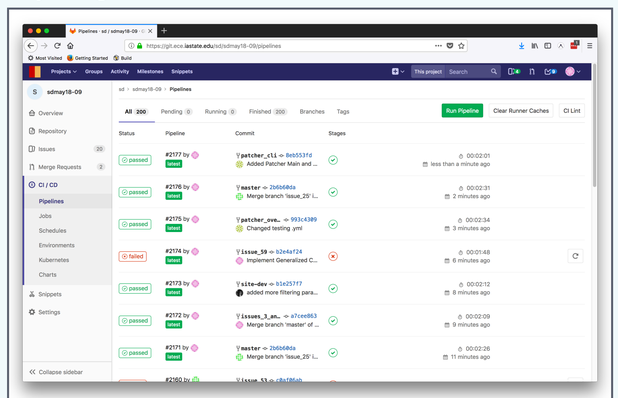

Figure 4: GitLab Continuous Integration Status Page

Because our pipeline is split into distinct modules with specific input/output value in mind, we focused on unit testing as a way to verify the stages of our pipeline worked correctly. Since the pipeline code is written in Java, we used the JUnit Testing Framework to create unit tests for all classes. We then set up a Continuous Integration (CI) system to automatically run these unit tests every time we changed the code. This was done using GitLab's built-in CI feature. We set up several Virtual Machines and Docker Containers to host a consistent testing environment, then connected these to our GitLab Repository. Each commit we made triggered the CI test suite to make sure our changes did not adversely affect the pipeline.

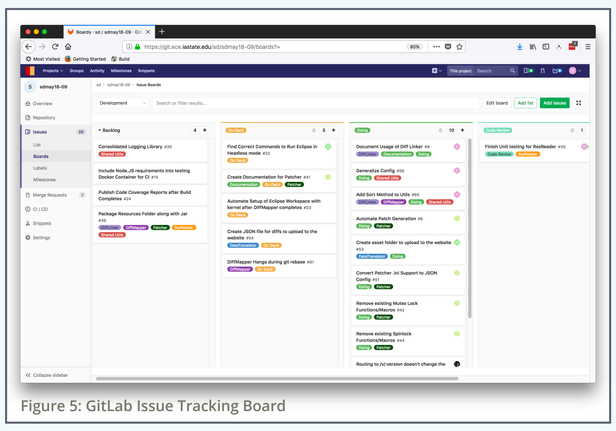

Figure 5: GitLab Issue Tracking Board

Once this CI system was setup, we then added a code coverage report to run after a successful run of unit tests. This allowed us to measure how much of the code base we were testing. We enabled checks in GitLab to only merge code changes to our master branch if all Unit Tests passed and our Code Coverage never decreased. Using this system, we were able to test 93% of our code base and can conclude that our pipeline runs correctly. In addition to the unit tests, we also performed several runs of the pipeline on a subset of the kernel. This was done to verify the correctness of data as we passed it between the different modules. This data has already been used by researchers to aid them in tracking instances. In a recent run, we compared a subset of versions 3.19-rc1 and 4.13 of the linux kernel. We were able to track ~2000 instances, and successfully link 700 of those instances. Of those, 700, we uncovered ~50 instances that were marked for further review by researchers since their pairing status changed. Tracking these "changed" instances hasn't been done before and will hopefully lead to some reports that can be submitted to kernel devel-

RELEVANT STANDARDS

IEEE 1233-1996: IEEE Guide for Developing System Requirements Specifications
IEEE 13210-1999: ISO/IEC 13210:1999, Information Technology -- Requirements and guidelines for test methods specifications and test method implementations for measuring conformance to POSIX(R) standards