Continuous Model-Based Verification of the Linux Kernel

Design Document

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

Revised: 4/25/2018 | Version 3

Table of Contents

1. Introduction **4**
- 1.1 Acknowledgement 4
- 1.2 Problem and Project Statement 4
- 1.3 operational Environment 4
- 1.4 Intended Users and uses 4
- 1.5 Assumptions and Limitations 5
- 1.6 Expected End Product and Deliverables 5

2. Specifications and Analysis **5**
- 2.1 Proposed Design 5
  - 2.1.1 DiffMapper 5
  - 2.1.2 Patcher 6
  - 2.1.3 Verifier 6
  - 2.1.4 DiffLinker 7
  - 2.1.5 Data Translator 7
  - 2.1.6 Website 7
- 2.2 Design Analysis 7
  - 2.2.1 Strengths 8
  - 2.2.2 Weaknesses 8
  - 2.2.3 Design Completion 8

3. Testing and Implementation **9**
- 3.1 Interface Specifications 9
- 3.2 Implementation Details 9
  - 3.2.1 RSS Reader 9
  - 3.2.2 DiffMapper 10
  - 3.2.3 Patcher 12
  - 3.2.4 Verifier 13
  - 3.2.5 DiffLinker 13
  - 3.2.6 Data Translator 14
  - 3.2.7 Website 14
- 3.3 Hardware and software 15
- 3.4 Process 15
- 3.5 Results 16
- 3.6 Functional Testing 16
- 3.7 Non-Functional Testing 16
- 3.8 Modeling/Simulation 17
- 3.9 Implementation Issues and Challenges 17

4. Closing Material **18**
- 4.1 Conclusion 18
- 4.2 References 18

1. Introduction

1.1 Acknowledgement

Our team would like to acknowledge Dr. Suraj C. Kothari and his PhD. student assistant Payas Awadhutkar. Both have provided and will continue to provide valuable technical advice and access to various resources that were crucial to the completion of this project.

1.2 Problem and Project Statement

Iowa State University's Knowledge Centric Software Lab (KCSL) has developed a tool[2] to verify the Linux kernel for specific types of bugs. Unfortunately, the tool requires a patch each time the kernel is updated, must be run manually, and cannot track changes to the Linux kernel between versions. Because Linux is used in many applications, it is important to find any bugs that could cause problems as quickly as possible.
Our team developed a system to automatically create a patch and run the tool every time the the kernel is updated. Also, as a way to make finding bugs quicker, we have created a website that posts the results of each run, and allows users to view and verify results. This output includes a way to view changes made to the kernel between versions for easier sorting.

1.3 operational Environment

Our product's operating environment is purely virtual since we have produced only software. Linux kernel verification takes place on a single machine and the results are displayed online. There is no physical product.

1.4 Intended Users and uses

Our intended users include, but are not limited to: The Linux kernel development team, software developers and researchers, and anyone interested in the integrity of the Linux kernel.

Our product only has one intended use: support and enhance the Linux kernel verification program our client has produced. This is achieved through an automated process that completes the following:

1. A patch is generated for the kernel that allows the verification program to run as intended
2. The verification program runs and generate data explaining what parts of the Linux kernel are verified safe and unsafe
3. This data is pushed to a server where users can view results and verify their accuracy.

1.5 Assumptions and Limitations

Assumptions

- Users of the website are willing and able to assist in kernel verification by viewing the instance data and comparing it to the next version instance data
- The verification program works as intended
- No sweeping changes to the organization of the Linux kernel will be made

Limitations

- The verification process must complete within 24 hours
- Web platform must be widely supported
- The verification program must be incorporated into the automation process

1.6 Expected End Product and Deliverables

1. Automated support for running the MBV toolbox on new versions of Linux. This includes the tool to create patches for the new versions, and the automation of the entire verification process. This also includes the tool to view changes to locking instances between kernel versions.
2. A public website to host the the verification evidence and facilitate collaboration for the verification process through analysis of the instance data mapped from our differencing algorithm.

2. Specifications and Analysis

2.1 Proposed Design

First, a system has been implemented to detect when a new version of the kernel is released. Using this system we can kick off a new run of our verification pipeline. The verification pipeline is split into several modules that run sequentially. They are the Diff Mapper, Patcher, Verifier (L-SAP), Diff Linker, and Data Translator. At the end of the pipeline, verification evidence and differenced results are generated and can be pushed to the website.

2.1.1 DiffMapper

The DiffMapper project aims map metadata between two versions 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 uses 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 are "mapped" and the kernel can be passed to the next step.

2.1.2 Patcher

Before L-SAP can run, a patch to the kernel must be created and applied. This patch contains redefinitions to all of the locking functions and macros that exist in the kernel. The Patcher generates a patch 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 then outputs the generated and modified files to a directory the user chooses.

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

2.1.3 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. This runs after both the metadata tags have been inserted and the patch has been created and applied. The results from L-SAP are then analyzed by later stages of the pipeline.

2.1.4 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.

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 reveal new insights into the kernel but are instead attempts to show researchers where to look for 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

2.1.5 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.

2.1.6 Website

The website is an endpoint for the researchers running L–SAP and for people who are curious about the security flaws in the linux kernel. It provides a way to sift through large amounts of data very quickly. Viewers are able to view the data in an intuitive, user-friendly way that makes sense to both the researchers and the common user.

2.2 Design Analysis

All of the core functionality of the modules has been developed and tested with various mock data. Further tests have been done with subsets of the kernel to make sure the results from one module are able to pass as inputs into another module.

2.2.1 Strengths

There are several strengths that allow the verifier pipeline to complete in a fast and efficient manner. The first is that all modules are kept separate and strict input and outputs are defined. This allows concurrent development of the modules, but also allows iterations for more efficient implementations without having to worry about messing up other parts of the pipeline.

Another strength is testing the pipeline. As discussed in the testing and implementation section, unit testing is an important role to making sure software functions properly. Splitting the pipeline into modules allows testing to happen mostly on a module level and ensure that modules work with expected inputs and outputs. This type of testing is faster and can cover specific edge cases for a module without disrupting other modules.

2.2.2 Weaknesses

There are some weaknesses with this design that focus mostly on maintenance and error handling. Because the modules are split up into separate project. This introduces more code to manage. Further, there are several functions that are similar and should be split in to shared library that all modules can be use. Such a shared library introduces dependency constraints for the modules.

Another weakness is with error handling. Because each module knows only its inputs and outputs, it is hard to handle an error in one module and still provide a reconciliation strategy to continue the pipeline. Because of this, if the pipeline fails, some analysis is required to determine where the error occurred and what steps are needed to prevent it from failing in the future.

2.2.3 Design Completion

The Rss Reader has been created and successfully provides notifications on a kernel update. This has not yet been linked to a full pipeline start, but instead requires a user to manually start the pipeline. This is for testing purposes, but will be connected in the near future to allow the pipeline to run automatically.

The Diff Mapper has been created and is able to successfully map 95% of a previous versions results into the kernel before L-SAP is run. This has been tested on several versions of the kernel and has been tested on jumps between multiple major and minor releases to ensure the solution works as the kernel changes a number of times.

The Patcher has been successfully automated and can generate patches that apply to the kernel after mapping is complete. This has not yet been connected to triggering L-SAP, but this connection can be made in the near future to allow the pipeline to run. This connection was not made to allow standalone testing of the patcher.

The Diff Linker has been created and is able to successfully link instances between versions and provide reports of instances that need further review. At this time, the only heuristic for marking an instances for further review is a change in status between versions, but this can be extended in the future if needed.

The Data Translator has been created and successfully converts result data into structured data that can be used by the website. The results on the current deploy of the website were generated using the Data Translator.

Finally, the website is able to meet the minimum requirements. Results data is displayed by version and can be filtered by a number of properties. This filtering happens reactively so the data set can be narrowed on a whim without significant time loss. Data view for instances shows all relevant info.

3. Testing and Implementation

3.1 Interface Specifications

Our project is focused specifically on software, but the verifier tool does require special hardware to run. There are machines available to us in the KCSL, however running the tool takes time. In order to work around this hardware and time requirements for testing, we decided to work with subsets of result data. These results were given to us from previous runs of L-SAP and allowed to compare the output of our modules with previous runs.

There are 4 major parts within the pipeline that we need to test. These include: 1) A patch generation system that analyzes the new version of the kernel for new changes and generates and applies the patch to the new version of the kernel; 2) A difference mapper that looks at the results of a previous version and creates a mapping to the new versions results; 3) A difference linker that uses the map to generate a list of new instances, removed instances, and changed instances; 4) a functioning website that can display the results of each run.

Some additional modules are necessary to make sure the automation between modules works properly. These sub parts include the RSS Reader and the Data Translator. All parts of the project include configuration options as an input to make sure the pipeline can operate in a number of situations. The outputs between modules are clearly defined, but vary for each module. See the later implementation details section for more information.

3.2 Implementation Details

3.2.1 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.

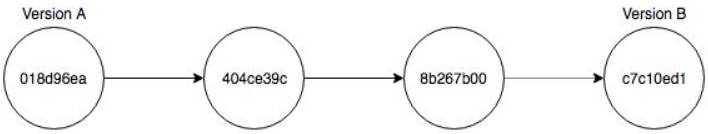
3.2.2 DiffMapper

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

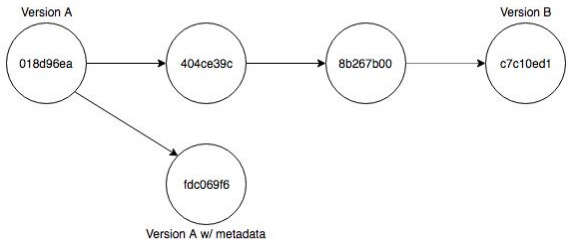
Once the kernel has been prepared with the old version, We track instances from the input instance map and store them in memory to be inserted in a batch based on the file name. This map is necessary from two perspectives:

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.

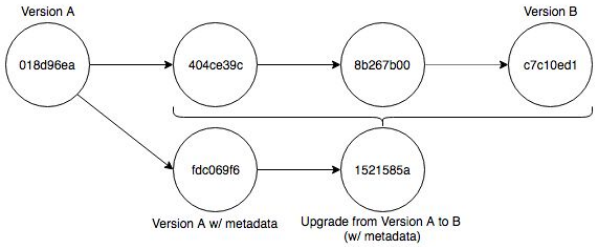


Use git to upgrade to the new version of the kernel

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.



3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

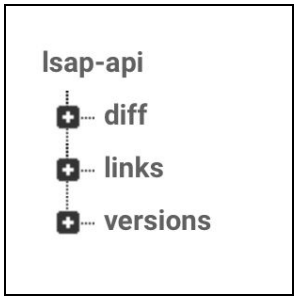
3.2.7 Website

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.



3.3 Hardware and software

We use a continuous integration system for running tests. We are currently looking at using the CI built in with GitLab[4]. The CI tool in GitLab was a natural choice for our testing process because we can automatically start the process based on changes to the source code. Finally, we can view the results within our development process, as well as require tests pass within the GitLab CI before allowing merges.

We use Java for the majority of our code, so we use JUnit to create unit tests. For the website, we use an angular framework with the main portion of the site written in Typescript. All of these unit tests can be run using GitLab CI, so this fits nicely with our testing process plan.

For the functional tests, we use a shell script to run these automatically. We have a version that we run with the mocked version of the verifier and one that uses the real verifier. As we stated previously, we run the mocked version to help develop the parts surrounding the verifier and once those parts are stable, we run tests using the real verifier.

3.4 Process

We test our code using continuous integration. Each pull request to our gitlab repository kicks off a CI process that is required to pass before the request is merged. We currently do this for every pull request from our remote repository. Since we are using GitLab's built-in CI process, we have provided a docker image that has eclipse and junit for java test, and node/npm for the javascript tests. We can then add/remove instances of these images automatically through Docker Machine.

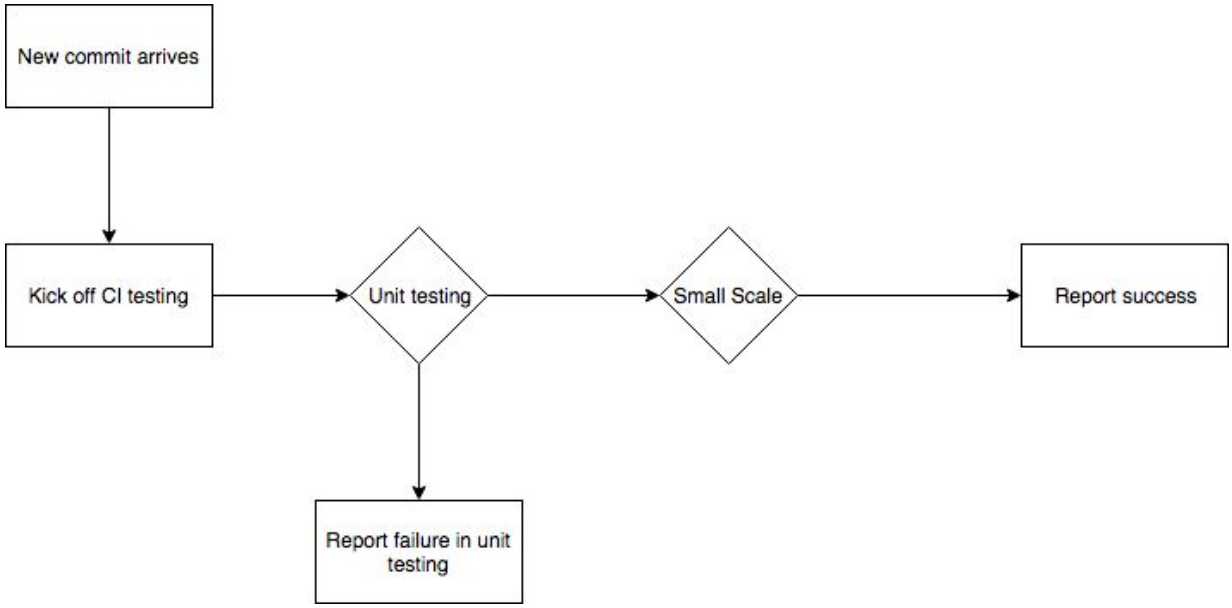


Figure 1.

3.5 Results

Using our tools, we have been able to generate a patch for kernel version 4.13. L-SAP was run on this patch, and afterwards we linked instances between 4.13 and the previous time the tool was run (version 3.19-rc1). Finally, we were able to translate the results of the run and post them to the website[5].

3.6 Functional Testing

Functional testing has been implemented via JUnit. Using GitLab's built-in continuous integration system, this test suite is run every time a developer pushes changes to our Git Repository. This ensures that any changes in the commits do not fail the tests. Merge Requests into the master branch are not allowed unless all tests pass.

This workflow ensures that the master branch is stable and all functionality we expect to work does work. Any new functionality added in a commit must also add tests to cover the new functionality.

3.7 Non-Functional Testing

In order to ensure the unit tests work properly and cover our code base, we generate code coverage reports for each module. This is then used to calculate an average coverage percentage. We use this to make sure our testing suite is testing the full functionality of our modules. This became another requirement for merge requests.

Performance Testing has been implemented for some modules to ensure they are able to handle different scales of data. This includes the DiffMapper and DiffLinker, since these modules deal directly with changing data. Various subsets of the kernel data have been created to test DiffMapper and DiffLinker at scale. This ranges from small subsets (10-50 instances) to the full set of the kernel (~20000 instances). These subsets have been created using 3 versions of the kernel. This is used to make sure the solutions are not version dependent.

Additional Logging is printed out for various modules which includes status messages and timings for various sections within a module. These are tracked and monitored to make sure the timing is consistent and there are no drastic increases in the time it takes to complete a module.

3.8 Modeling/Simulation

We have generated test data that mimics inputs and outputs of L-SAP. This was used in our tests to make sure our modules function properly. As stated earlier, we also took previous results from L-SAP and created subsets of the data in order to test our modules at different scales.

3.9 Implementation Issues and Challenges

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.

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.

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.

4. Closing Material

4.1 Conclusion

Our final project supports and enhances the product our client has created to verify the integrity of the Linux kernel, and shows evidence of discovered bugs on a easy to access webpage. Our solution is modular, and each module handles a specific part of the kernel verification pipeline: new release detection, difference mapping, patching, verification, difference linking, data translation, and web-based data display. We have created these modules based on the needs and advice of our client.

4.2 References

[1] Project proposal document: http://bit.ly/2hoSuRO
[2] L-SAP tool: http://home.engineering.iastate.edu/~atamrawi/l-sap/index.html
[3] Git Documentation: https://git-scm.com/docs
[4] GitLab CI Documentation: https://docs.gitlab.com/ee/ci/
[5] Website Displaying Results: https://lsap.knowledgecentricsoftwarelab.com/v/413