Continuous Model-Based Verification of the Linux Kernel

Design Document

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

Revised: 12/3/2017 | Version 2

Table of Contents

1 Introduction

1.1 Acknowledgement

Our team would like to acknowledge Dr. Suresh 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 will be 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, in its current state the tool requires a patch each time time the kernel is updated, and must be run manually. Because Linux is used in many applications, it is important to find any bugs that could cause problems as quickly as possible.

Our team will develop 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 will create a website that will post the results of each run, and allow users to view and verify results. This output will include 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 will be producing only software. Linux kernel verification will take place on a single machine and the results will be displayed online, but there is currently no plan for a 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: verify the integrity of the Linux kernel. This will be achieved through an automated process where a patch will be generated for the kernel that will allow the verification program to run as intended, the verification program will run and generate data explaining what parts of the Linux kernel are verified safe and

unsafe, and this data will be pushed to a server where users can view results and verify their accuracy.

1.5 Assumptions and Limitations

Assumptions

- Users of the website will be willing and able to assist in kernel verification
- 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 provided 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. We will also determine which control paths have changed, and only consider those for viewing.

2. A public website to host the the verification evidence and facilitate collaboration for the verification process.

2. Specifications and Analysis

2.1 Proposed Design

First, we will implement a system that will detect when a new version of the kernel is released. Using this system we can kick off a new run of our verification pipeline. To run the verifier tool, we will first automatically generate a patch that will enable the tool to work with the new version, then we will have to apply said patch to the kernel. Next, we will start the tool on the patched kernel, and once that is done will need to run the results through a differencing program so that we know what has changed in this version. Finally, we will upload the results to the website. These steps will most likely be implemented within an Eclipse plugin so that we can interface with the verification tool, which is also a plugin for Eclipse.

The website will allow users to see the latest results from the verification tool. We will store the results in a database, with a unique identifier for the images associated with them. These identifiers will allow us to setup a "search" function on the site to help filter the locks based on things such as: type of lock, location, etc. This is wrapped inside of the minimum viable product that the site will function as. Further functionality, and some of our stretch goals will include the addition of users to the site. Users will be able to submit feedback to certain pairings (or even inconclusive pairs) and graph data to provide a backup verification. Also, a feature we would like to have is a way to compare two or more versions of an instance. And finally, we will add in a tool to interactively view instance graphs.

2.2 Design Analysis

So far we have manually applied a patch to a newer version of the kernel. We tested this by verifying the kernel compiled with the new patch. This will allow the tool to run on that version, and helped us to understand the creation an application of the patch so that we will be able to automate it in the future.

We have also begun to create a tool that will be able to diff the results of new runs of the tool compared to previous ones. The current prototype can map instances of code between versions, an important first step to finding the differences. We have tested this between two instances of the tool's runs, and verified it produces the correct results.

We have begun to implement the website to meet the minimum requirements. These requirements include the filtering and reorganizing of results from the tool. There is a website design that is currently being used, however it provides no search functionality and does not allow the user to find instances of locks easily. Since viewing the graphs/data about the different locks in an organized way is highly important to our stakeholder, we have decided to start on a new design that will meet the needs while still being able to handle many instances.

So far a major strength of our approach is the how we separated it out into several modules. This means the code is flexible and opens us up to more options for deployment. This also allows us to create a pipeline of modules, and could run multiple versions of the kernel running on separate parts at the same time if releases come out close together.

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 requirement for testing, we have decided to build a mocked version of the verifier that can be used in place for

testing purposes. This version will take the correct input in, but will return a specific response without running through the full process of the tool. This will only apply to incremental changes of our pipeline in order to speed the rate of development.

There are 4 major parts within the pipeline that we will 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 differencing engine that uses the map to generate a list of new instances, removed instances, and changed instances; 4) a functioning website that will be able to view the results of each run.

Each part will have its own group of unit tests that need to be run and all of this can be automated through a continuous integration system. Additionally, we will be constructing functional tests that act as "dry-runs" of our system to run through the pipeline within tested versions and make sure the correct results get generated. These tests will walk through the pipeline with the mocked version of the verifier. Our plan is to start with reduced versions of the kernel to limit the number of instances we have to test, then use full versions of the kernel to make sure our pipeline can handle the scale. When these tests pass, we then plan to run the same tests using the real version of the verifier. Each run of tests using the real verifier will generate the necessary artifacts we need to deploy to the website, so we can then perform a deploy after each successful run of the full pipeline.

3.2 Hardware and software

We plan to 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 will be using Java for the majority of our code, so we will be using JUnit to create unit tests. For the website, we will be using a framework in JavaScript (such as React or Angular) and can create with that framework. All of these unit tests can be run using GitLab CI, so this will fit nicely with our testing process plan.

For the functional tests, we plan to use a shell script to run these automatically. We will 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 plan to run the mocked version to help develop the parts surrounding the verifier and once those parts are stable, we plan to run tests using the real verifier.

3.3 Process

We will test our code using continuous integration. Each pull request to our gitlab repository will kick off a CI process that will be required to pass before the request is merged. Eventually, we plan to scale this to every commit, but starting on every pull request will allow us to check how the CI pipeline is working before scaling it. Since we are using GitLab's built-in CI process, we will provide 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.

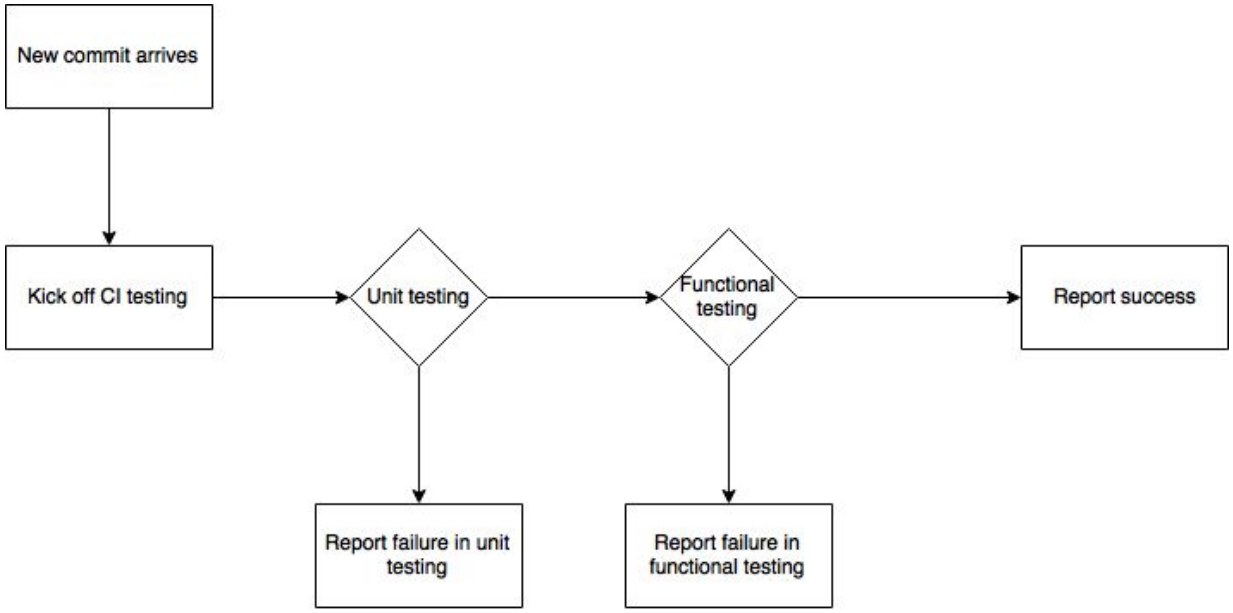

Figure 1.

3.4 Results

We are still in the process of setting up our testing infrastructure, and so far we have no results to report from unit tests.

As stated previously (Section 2.2), we have been successful with manual tests of both the patching and instance mapping systems. This includes creating a patch manually for multiple versions of the kernel and verifying that the kernel builds successfully. We plan to use these versions as test cases and compare the results generated automatically to the manual versions generated. We also plan to build the kernel for the automatically generated patches in order to verify build correctly. The instance mapping was tested using a reduced test case that we plan to transfer to a unit test. We are currently in the process of testing the full kernel, and that will be transferred to a unit test as well.

3.5 Functional Testing

We plan to implement our functional testing via JUnit, and using continuous integration to make sure every change we make is up to spec. These test will ensure that our tool will match the specifications laid out by our client. This will include testing the correctness of our patching algorithm, our differencing algorithm, and other smaller parts of the project.

3.6 Non-Functional Testing

We plan to implement both Resiliency testing and Performance testing. For resiliency testing, we will run our solution on many versions of the Linux kernel, including older versions. This will help us determine if our solution is applicable to all versions of the kernel or if it applies only to the version we used when creating the solution. For Performance testing, we need to ensure that our solution completes in a timely manner and uses as few computational resources (such as RAM and CPU utilization) as possible. The entire pipeline needs to complete within eighteen hour period, and L-SAP currently takes about twelve hours to complete when it has access to all of the computer's resources. We need to keep the computation time of our solution, which is the remainder of the pipeline, under six hours maximum, but would like to keep the time under ten minutes.

3.7 Modeling/Simulation

We will need to simulate the running of L-SAP in many of our test cases. This is because the time and resources L-SAP will take to run are infeasible for running our tests.

4 Closing Material

4.1 Conclusion

Our final project will provide a way to find bugs in the Linux kernel automatically, and show evidence of the bugs on a easy to access webpage. So far we have begin to create modules to help create this project such as applying a patch to the kernel, designing a diff tool to find differences between runs of the tool, and beginning to design the website. We will continue to develop these and other modules important to the final design as described by 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/