Automated Regression Hunting

PyReT and the Linux kernel

Aaron Bowen
Neumont University
abowen@student.neumont.edu

Paul Fox
Neumont University
paul.mf@gmail.com

James M. Kenefick, Jr.
IBM
jkenefic@us.ibm.com

Ashton Romney
Neumont University
Ashton.Romney@hotmail.com

Jason Ruesch
Neumont University
jason@jasonruesch.com

Jeremy Wilde
Neumont University
jwilde@student.neumont.edu

Justin Wilson
Neumont University
jwilson@student.neumont.edu

Abstract

Code regressions happen, it is almost a truism, and the more complex the system, the more often they will occur. Detecting which code addition or patch created the regression, while sometimes time-consuming, is at heart an iterative process which lends itself to automation. The core of this process is the same regardless of the kind of software being written, whether it is application programming or operating system programming.

The PyRet project (http://pyret.sf.net) has automated this process in an object-oriented fashion using Python. It includes an implementation to test the Linux Kernel using the Linux Test Project (http://ltp.sf.net). This paper will describe the process of regression hunting which lies at the heart of the PyRet project. It will provide: a basic overview of the PyReT architecture, the search models that PyReT currently utilizes, and methods for extending the PyReT tool to other software projects.

1 Introduction

The purpose of this paper is to describe the process of automated regression identification. It will focus primarily on automation of identification of patches which cause regressions within the Linux kernel. It will describe the implementation of automated regression detection by the PyReT project (http://pyret.

`sf.net`). It will cover searching of the regression space, hunting the regression with multiple systems, and the communication solution selected for managing multiple systems. It will detail the specifics of hunt configuration within PyReT; i.e., the mechanisms for managing the kernel build and test phases of the regression hunt. Finally, it will cover extension of the tool to other non-kernel based testing and what still needs to be done going forward with PyReT.

2 Regression Hunting

2.1 Overview

Software regressions are bugs existing in a specific version of software that are not present in previous versions. Knowing which specific update introduced a given bug can drastically affect the amount of time needed to address the issue. The regression hunting process can be extremely time-consuming and error-prone. This process is fairly standard regardless of the development environment and lends itself very well to automation. By automating the regression hunting process, developers are able to spend their time fixing bugs rather than tracking down where they were introduced.

2.2 The Process

The first step in any regression hunt is to determine the range of change-sets or dates that will be searched. This step is crucial to the process because the size of the initial range affects the number of tests which must be performed to determine which update introduced a given regression. This range should start with the first change-set where the bug is not present and end with the first change-set where the bug is known to exist. The smaller the range, the faster the search will be.

The second step in the regression hunting process is to specify the test case to be performed which will determine if the regression is present in the current build. This should be fairly straightforward—as the regression manifested somehow, the test case should just re-create the activity that manifested the regression. Next, the development environment is loaded with a given change-set, the test case is checked and passes, fails, or returns an error code based on whether the regression was detected within the current build.

The last step in the regression hunting process is to analyze the results of the first two steps and decide what needs to happen next in the hunt. The system needs to determine if the hunt needs to continue or if it has completed and a single change-set or small selection of change-sets has been identified as having introduced the regression.

2.3 Challenges

Automated regression detection, while fairly straightforward, can present several challenges such as intermittent regressions, branched development environments, efficient hardware utilization, and of course the inevitable unautomatable test case.

Intermittent Regressions are bugs that inconsistently return pass or fail values when checked against your test case. In situations where intermittent regressions are encountered, it may not be possible to find the exact change-set that introduced a given regression. Hunting for intermittent regressions can still give developers valuable information that can be used to narrow the range of change-sets that they need to manually check.

When searching for regressions in development environments that use branching, determining the starting range of change-sets to check becomes a bit more difficult. A given regression can exist on one branch and not on another. A regression could have been corrected on the main branch and then re-introduced from a merged development branch. Regression-hunting algorithms must be able to correctly handle these situations.

One of the greatest benefits of automating regression detection is the ability to take advantage of multiple machines to test more than one change-set at once. This has the potential to drastically decrease the time needed to search a large range of change-sets when a single test takes a great deal of time to complete. This feature introduces challenges of its own. It is now necessary to develop regression-hunting algorithms that can take advantage of the multiple hunter machines instead of using a simple binary search.

And finally there will always be test cases that cannot be automated—kicking out a power cable, pulling a raid disk—and sometimes the effort and resources required to automate the test case just aren't worth it.

3 The PyReT Solution

3.1 Overview

As regression hunting is at its heart an iterative process, PyReT was designed to implement this process, rather than to solve a specific regression problem.

The solution was designed as two major pieces. A Hunter, which executes a given test in the search space, and a Master, which has responsibility for managing search spaces. The application flows of the Hunter and Master were mapped out based on those areas of responsibility. Next the individual tasks of the application flows were categorized and encapsulated into objects in the form of Python classes. This will allow specific implementations to customize the process as needed, but allows the common portions to be reused without change.

The modules used for a specific hunt are configured in a search definition. The Master and Hunter applications use the definition to establish the boundaries of their regression and determine the appropriate application code and test cases for the current regression under investigation. It is the responsibility for the end user to provide the search definition for a hunt.

The Master module uses the search definition to create the search space and from that test jobs are created which are then executed by Hunters, and the results reported back to the Master module which uses this information to further orchestrate the hunt.

All of this application interaction is managed by the communication subsystem, which not only owns the responsibility for facilitating application communication but also owns the logging mission. The communication subsystem is also implemented as a set of modules to facilitate customization to specific environments.

3.2 Application Flow

There are three primary application work flows used by PyRet:

- The Master control process, which is responsible for the overall management of all the search spaces currently under analisys.

- The Master search process, which owns a single search space and is responsible for identifying the patch which caused the regression.
- And the Hunter work flow, which owns the execution of a single instance of the regression test against the patched application under test.

The Master control process workflow starts by loading the Master config and confirms the availability of the communication subsystems. Next the process kicks off an infinite loop referred to as the job thread. This loop basically spins looking for new search definitions. When a new definition is identified a Master search process is initiated.

The Master search process begins by processing the search space, identifying all patches and patch set boundaries. Next it loads the regression search module, which handles how the search space is to be traversed. Jobs are then created, executed (by the Hunter), evaluated, and new jobs created until the regression is identified or the search space exhausted.

The Hunter process spins on the communication subsystem until it finds a job it is designated to execute. Once a job has been identified, the Hunter will obtain and patch the application. Next the application will be installed and initiated on the Hunter system and the regression test will then be executed against the target application. Finally the results of the job will be passed back to the Master search process via the communication subsystem.

3.3 Current Searches

Once a set of change-sets has been decided upon for testing, an efficient way to quickly test the range of change-sets must be implemented to ensure the efficiency of PyReT. The binary search was originally chosen for its simplicity, ease of initial design, and efficiency. Currently there are two search modules included within the PyReT framework, a single-hunter binary search and a multi-hunter search.

After a change-set range has been determined the patch that lies in the middle of the range will be sent off as the first job for a hunter to build and test. The direction to travel after the first job is based upon its results: *pass*, *fail*, or *error*. If the patch has passed then it is known that the regression, if any, lies somewhere between this patch and the ending patch within the change-set range. If the patch has failed then it is known that the regression lies somewhere before this patch if the regression is within the range of the change-sets being tested. This process continues until we are left with a *pass* patch jointly next to a *fail* patch, and this *fail* patch is recorded as introducing the regression.

In a perfect world a passed patch and a failed patch would be exactly what one would want in the entirety of their results. Unfortunately we are not in a perfect world and must consider the possibilities of unforeseen events, which is the reason for the *error* result. If for some reason a hunter is taken offline, crashes, exceeds the time-limit set forth for the patch, etc., then the job is neither *pass* or *fail* but is of type *error*. In this case the current patch is set as a temporary boundary and the next patch tested is taken as if the current one had passed. This process continues on until a result is found, a *pass* patch immediately next to a *fail* patch. If a result is not found then sub-searches are created, testing in-between each error until a result is discovered or the entire patch has been tested without a conclusive result.

The single hunter binary search relies on a single hunter to perform all of the work required by a given range of change-sets. Since

there is only a single hunter being utilized then only one patch can be tested at any given time throughout the range of the change-set. Once a result has been obtained the next patch is sent out to the hunter and a result is waited upon, and so on until the test has completed.

The multiple hunter search can be far more efficient than that of the single binary search, where there is an extensive range of change-sets. First, before sending out patches to be tested, the number of available hunters is used to determine how many patches will be tested initially. Once this number is acquired patches are sent out to available hunters to be tested; these patches are evenly spaced throughout the entire range of the change-set. This can potentially find the regression within the change-set exponentially faster than that of the single-hunter binary search. Once this chunk of patches has been tested the new boundaries are set, the number of available hunters is acquired, and the new patches are sent to available hunters, spread evenly across the new range of change-sets to be tested. This continues, like above, until a regression has been discovered, or an inconclusive result is determined.

3.4 Communication

PyReT relies on a shared files system (SFS) for all communication between systems involved in the hunting process. This allows the code within the applications to be kept very simple as compared to using a direct network connection. The directories that exist here fall into one of three categories: search setup, communication, and logging.

The search setup directories are where search space definitions are looked for and stored on completion. This location is routinely scanned by the Master to detect when a new one has been created and it will then kick off the search process previously described. This is also where the source code and patch staging occurs.

The communication directories are quite active. Any time a Hunter is started it will create a file in this area to notify the Master that it is available. This file is also used to indicate a Hunter's current state by altering its extension. The Hunter is also regularly touching these files to let the Master know that they are still processing. This area is also where a Hunter looks to see if it has been assigned any jobs. This is accomplished by the Master by creating a job file and setting the extension to the name of the Hunter that it is being assigned to. On completion of a job the hunter will change the extension to indicate the result. There is also a location where all the completed jobs are moved once all processing is completed.

The third category of usage for the SFS is where results are also stored. Each system reports on what it is doing. These directories are most useful while debugging a new implementation or gathering information about hunts that were inconclusive.

3.5 Module Base Classes

In order for PyReT to effectively orchestrate the activities of the disjointed module implementations it has certain expectations of how they will behave and the kinds of tasks they are expected to carry out. The messages they must respond to are defined in the project, the expectations for behavior are outlined below.

The Transport module deals with acquiring source trees and is Master specific as are the next three modules. It is called only once when a search is first started. During this invocation it should make sure that the copy of the source tree that the master will be working with

is at least current enough to cover the range of change-sets specified in the configuration.

Once the source code has been updated, the Decompress module is called. If the source tree is not in a compressed form this module can be omitted or the DecompreNone version can be used. If the source was transported in a tar ball or some other compressed format, it would be the responsibility of this module to extract the individual files.

In order to make sense of the source tree the Master relies on the MasterPatchSource module. It will call this module's SplitChangeSetIDs which is expected to return a list of identifiers that can later be used to patch the source to that revision or copy the correct version. It is also expected that at the end of this process the hunter will only need the change-set id to patch the source it is working with. As an example the Linux Kernel implementation creates a patch file using this id as its name in this method.

The RegressionSearch module is what determines which change-sets will be tested and in which order. When it is created it is passed the change-set list and is expected to retain which tests were run and the results of each test. The module is asked for a batch of indexes to be tested, the Master will then issue a job for each set. As each test is completed it will notify this module, once they are all complete, if the module has reported that the search is not complete it will request the next set to test. Once the hunt is finished it will request the results list from the module.

The first of the Hunter-specific modules, Copy, deals with copying the source from the shared file system to the Hunter's local file system. It is a basic module expecting only to know where it should copy the source from and where to place it.

The implementation of the HunterPatchSource module is very dependent on the behavior of the MasterPatchSource module. This one is expected to apply the patch to the copied source to bring it up to the revision that it is to test. This module could be empty if the source files were split up by revisions so that the Hunter could just copy the correct version.

The Build module is one of the straightforward pieces. Assuming all the previous steps have worked, it is tasked with compiling the source code in preparation for the test.

In order to set up the compiled source, the Hunter will call the Install module. Here any configuration of the system should occur. This module is also tasked with undoing any changes made to the system and will be notified to do this once the test is completed.

The Test module is called to exercise the code under review. It is expected to return the result of the test to the Hunter so that it can be communicated back to the Master.

3.6 Module Configuration

The PyReT application uses the concept of encapsulation, implemented as Python modules, as a way of making it versatile. Each piece of functionality that does something towards finding the regression is placed into a module. The modules are then loaded and used by both the Master and Hunter applications.

This first section discusses how a module is defined. There is a configuration file, referred to as the search space file, which contains the definitions for the modules. Each module can contain parameters. The developers of the module decide what parameters the module accepts or requires. They can then document how their module is to be defined in the search space

file or, if they want, they can define a special method in their module file that gives the caller the parameters needed by their module.

The search-space file currently being used by PyReT's implementation of finding regressions in the Linux kernel contains the following modules: BinaryRS, MultiHunterRS, and CompleteRS regression search modules; TransportCogito transport module; DecompressNone decompress module; LinuxKernelMasterPatchSource and LinuxKernelHunterPatchSource patchsource modules; LinuxKernelCopy copy module; LinuxKernelBuild build module; LinuxKernelInstall install module; and LinuxKernelTest test module. A module is either required or optional. For instance, a search module is required, or the application would be useless. In contrast, the decompress module is optional because the program being tested may not need to be decompressed in any way, as is the case with the Linux kernel implementation. It does, however, make use of the DecompressNone module in order to showcase that there can be a decompress module, if needed.

The modules to be used and the parameters they will receive are specified in the search-space file. Creating this file is a time-consuming process when done from scratch. To write one, the name of every module being used and what parameters each takes needs to be known. A sample file is provided, but it also requires considerable knowledge of what modules exist in order to adapt it. The search-space creation program will dynamically read what modules are available and prompt the user to choose which ones to use. For each module being defined, it will gather what parameters the module is expecting from a global getParameters method defined in the module file by the developer. This makes it much easier to accurately specify a new search space.

4 The PyReT Exemplar

4.1 The Linux Kernel

PyReT could have been used to test other projects; however, there are a number of characteristics that make some projects easier to work with. These key aspects would be an open repository that can be manipulated via automation, and the availability of tests to run against that source. We found these properties readily available for the Linux kernel in the form of GIT and the Linux Test Project. In this implementation, Cogito is utilized to facilitate all interactions with GIT.

The interaction with the source tree is driven by the Master. It will use the search definition to update or create a local copy of the source tree each time a new search is started. It then pulls the log for the specified date range and parses to determine which change-sets were committed during that time. This list is used by the regression searches to identify change-sets to test. Within the patch source module, which is also specific to Cogito, it uses the change-set list to revert the working tree of the cloned repository to the oldest revision in the list. It then iterates through the list, creating a patch file for each revision that might be tested and saving it to the shared file system. At this point in the process the Master has everything it needs to run the complete search. It then issues a job to a Hunter with the information of which change-set it is to test. With the initial setup it becomes a straightforward task of copying the working tree and applying the appropriate patch file from the shared file system and then executing the tests.

4.2 Handling System Restarts

The Hunter software runs as a daemon to allow it to execute at system startup and attempts to

register itself to do just that each time it starts. To deal with the problem of losing state between restarts, it saves its working directory in a fixed location to allow this to be reloaded. It also stores information about which step in the hunting process it was at so that it can pick up where it left off. The hunter also manipulates the boot configuration to make the newly built kernel the one that will be loaded by default. Upon completion of the tests it will revert this configuration to its previous state.

4.3 Testing with the LTP

The idea of a regression is that something that used to work, now for one reason or another, no longer does. Most regressions normally seem to be found by end users when the application, OS, etc. explodes, taking some bit of important work with it, but on occasion and depending on the application a regression bucket may exist, and in fact for the Linux kernel this is the case, The Linux Test Project (LTP). The LTP consists of over 2900 tests for exercising the Linux kernel, and executes (depending on who you ask, what you compile in the kernel, the phase of the moon) about thirty to forty percent of the kernel. This combined with the ability to execute a single test case at a time made it a good exemplar to use for this project. It is, however, pretty important to note that any test case which can be automated can be used in place of the LTP.

Basically, once a test has been identified, create a test module from the PyReT class Test, which wraps the test. This module should do any prep for the test (pull the test code if needed, path setup, envrionment variables, whatever), execute the test, and return whether the test passed or failed. This module will then need to be registered with the system, after which it will be called as part of the regression search.

5 Extending PyReT

The modular design of PyReT makes it highly customizable. This is accomplished by implementing the base classes that are defined to cover each of the identified steps in the process. Included in PyReT's documentation is a set of HTML PyDocs describing these classes. With some effort, a developer can adapt PyReT to fulfill the needs of most projects.

It appears the most difficult part of this process is the creation of the modules dealing with the source code repository. The modules impacted by this are Transport and MasterPatchSource. The Transport module is a straightforward implementation, but it is very specific to which source control system in use. These modules will be highly reusable, as simple parameters of where to acquire updates to the source is all that will differentiate them between different applications. The MasterPatchSource module is more involved. It is tasked with identifying which change-sets exist in the date range to be tested. It is also responsible for making available the oldest revision and patches from that revision to each later revision to be tested.

It is expected that most of the other module implementations will be comparatively easy. In the case of the RegressionSearch module there is no need to define a new one in order to make use of PyReT, as the current implementations have no reliance on how the other modules function. The others will usually involve interacting with `stdin` and `stdout` to start and monitor each step of the process.

Any new modules are expected to be in the same directory as the base they are inheriting from. This will allow the Master to locate them when they are referenced in a search space file. Optionally a module can also define a global method that defines the parameters it is looking for. This will allow the search space definition

tool to prompt users for these parameters when they select a module with those definitions.

6 What's Next?

PyReT is still a young project and has a lot of room for growth and improvement. Key features for future development include a web interface and advance search modules.

Web interface—Currently it is difficult to setup a search-space with all of the necessary parameters to perform a regression hunt with the correct modules. A web interface for setting up regression hunts and reviewing the results of searches would allow users to easily select the type of regression hunt to perform as well as guiding them through parameter selection, ensuring a properly configured hunt.

Advance search modules—Search modules need to be developed which allow searching for multiple regressions simultaneously. Future search development should focus on new regression search algorithms that will make much better use of the distributed computing aspects of the PyReT project.

Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin