Augmenting Traditional Static Analysis With Commonly Available Metadata

by

Devin C. Cook

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 10, 2015

Keywords: data mining, computer security, vulnerability detection, static analysis, information assurance, metadata, version control

Copyright 2015 by Devin C. Cook

Approved by

John A. Hamilton, Jr., Chair, Alumni Professor of Computer Science and Software Engineering
Munawar Hafiz, Assistant Professor of Computer Science and Software Engineering
Jeff Overbey, Assistant Professor of Computer Science and Software Engineering
David Umphress, Associate Professor of Computer Science and Software Engineering

Abstract

Developers and security analysts have been using static analysis for a long time to analyze programs for defects and vulnerabilities with some success. Generally a static analysis tool is run on the source code for a given program, flagging areas of code that need to be further inspected by a human analyst. These areas may be obvious bugs like potential buffer overflows, information leakage flaws, or the use of uninitialized variables. These tools tend to work fairly well – every year they find many important bugs. These tools are more impressive considering the fact that they only examine the source code, which may be very complex. Now consider the amount of data available that these tools do not analyze. There are many pieces of information that would prove invaluable for finding bugs in code, things such as a history of bug reports, a history of all changes to the code, information about committers, etc. By leveraging all this additional data, it is possible to find more bugs with less user interaction, as well as track useful metrics such as number and type of defects injected by committer. This dissertation provides a method for leveraging development metadata to find bugs that would otherwise be difficult to find using standard static analysis tools. We showcase two case studies that demonstrate the ability to find 0day vulnerabilities in large and small software projects by finding new vulnerabilities in the cpython and Roundup open source projects.

Acknowledgments

I would like to thank my wife, Audra, for being kind and supportive throughout this process. I would also like to thank my advisor, Dr. Hamilton, and the rest of my committee for being helpful and understanding. I would like to thank Dr. Bob McGraw and Dr. Richard MacDonald at Ram Laboratories for allowing us to work on a related project that helped provide some relevant static and dynamic analysis experience.

I would like to thank Dr. Christopher Harrison, Dr. Christopher Perr, and the rest of the lab for working with me the past few years and allowing me to bounce ideas off of them.

This research was funded in part by Sandia National Laboratories[1], via a project under the supervision of Dr. Elisha Choe.

Finally, I would like to thank supernothing for giving me the inspiration for this project and helping me brainstorm.

[1]Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

Table of Contents

Abstract . . . ii

Acknowledgments . . . iii

List of Figures . . . vi

List of Tables . . . vii

1 Introduction . . . 1

1.1 Background . . . 2

1.2 Inspiration . . . 3

2 Survey of Literature . . . 6

2.1 Vulnerability Detection . . . 6

2.1.1 Workflow . . . 7

2.1.2 Summary . . . 11

2.2 Existing Vulnerability Discovery Tools . . . 12

2.2.1 RASAr . . . 12

2.2.2 rosecheckers . . . 12

2.2.3 BitBlaze . . . 14

2.2.4 ISA . . . 16

2.2.5 MAYHEM . . . 17

2.3 Leveraging Development Metadata . . . 18

2.4 Manipulating Source Code . . . 19

2.4.1 Text Matching . . . 19

2.4.2 Token Matching . . . 20

2.4.3 Matching Abstract Syntax Trees . . . 21

2.4.4 Refactoring Tools . . . 23

3 Experimental Design 25
3.1 Architecture 27
3.2 Data Gathering and Organization 31
3.3 Bug Correlation 31
3.4 Bug Injection Identification 33
3.4.1 Find Injection Using VCS Annotation 34
3.4.2 Find injection using added test cases and VCS bisection 35
3.5 Find New Bugs 37
3.5.1 Text Matching 37
3.5.2 Machine Learning 38
3.6 Manual Verification 40
3.7 Resolve Vulnerabilities 41
4 Results and Validation 42
4.1 Python Results 43
4.2 Roundup Results 47
4.3 Vulnerability Injection Patterns 49
4.4 False Positives 51
4.5 Summary 53
5 Conclusions 54
6 Future Work 57
Appendices 67
A Code Listing 68

List of Figures

2.1 Implementation of the Euclidean Algorithm 7

2.2 An Example Abstract Syntax Tree 8

2.3 Clang Static Analyzer 9

2.4 RASAr Architecture 13

2.5 Vine Architecture Overview 14

2.6 ISA Architecture 16

3.1 Project Architecture 28

3.2 Local Cache/DB Schema 32

4.1 Time To Fix Security Issues in Python 48

4.2 Time To Find Security Issues in Python 49

List of Tables

4.1 Python Developer Vulnerability Injection Rate (Top 10) 45

4.2 Python Developer Vulnerability Reports (Top 10) 46

4.3 Python Developer Vulnerability Resolutions (Top 10) 47

4.4 Files in the Python Project Containing Security Issues (Top 10) 50

4.5 Vulnerability Detection Rates 52

Chapter 1

Introduction

This dissertation showcases some of the benefits of using common data mining techniques to augment traditional static analysis. By leveraging the wealth of information provided by a project's bug tracker and source code repository, we can find bugs that traditional static analysis tools may miss. Static analysis has been used for a long time to analyze programs for defects and vulnerabilities with some success. Generally, a static analysis tool is run on the source code for a given program, flagging areas of code that need to be further inspected by a human analyst. These areas may be obvious bugs like potential buffer overflows, information leakage flaws, or the use of uninitialized variables. These tools tend to work fairly well – every year they find many important bugs. These tools are more impressive considering the fact that they only examine the source code, which may be very complex. Now consider the amount of data available that these tools do not analyze. There are many pieces of information that would prove invaluable for finding bugs in code, things such as a history of bug reports, a history of all changes to the code, information about committers, etc. By leveraging all this additional data, it is possible to find more bugs with less user interaction, as well as track useful metrics such as number and type of defects injected by committer.

Through two case studies using test data from cpython and Roundup, I will demonstrate that it is possible to use existing development metadata for an application to identify new vulnerabilities that are similar to previously reported vulnerabilities within that application, that it is possible to then use those learned signatures and models to identify new vulnerabilities in an unrelated application, and that it is possible to calculate some interesting and useful development metrics that may assist future development of secure code.

In order to see the benefits of this approach, it is important to first gain an understanding of the existing tools, and learn about other similar approaches that are used by security analysts when working with and investigating both vulnerable and malicious code. There has been much work in the fields of both static and dynamic analysis, but adding this additional layer of information in the form of development metadata yields important gains when working to defend potentially vulnerable software in use in production environments. The current approach mostly neglects this metadata.

1.1 Background

Static analysis is already a useful tool in the security world, however, it is generally used by a skilled security analyst while performing an application security assessment. It is but one piece in a larger security assessment process.

There are some detractors that claim static analysis is not enough to get a picture of what the code actually does. They actually have a valid point; it is very hard to tell with great certainty how code will execute unless it is actually being actively executed. [26] It is important for security practitioners to keep this fact in mind when trying to secure an application. Security is a process, and it is important not to rely on one singular type of tool.

Still, we must consider the amount of low-hanging fruit, and writing a tool to automatically perform static analysis is much easier than writing a tool to perform dynamic analysis of a binary executable. Simple debugging is relatively easy compared to the task of analyzing security implications dynamically.

For confirmation of this fact, simply inspect the state of the antivirus business. In the academic world, it is widely agreed that heuristic antivirus (antivirus that works by looking for suspicious patterns of activity) is theoretically better than signature-based approaches (that compare file signatures to a blacklist) [51] [29], but virtually every single antivirus

product on the market today uses a primarily signature-based approach. In fact, even application whitelisting, which is much easier to implement than a full-on automated dynamic analysis framework or even just a simple heuristic approach, has not become popular.

The simple truth is that any effort, no matter how minuscule, would be a vast improvement over the current situation. If it is possible to find just one previously unknown bug (these bugs are referred to as "0days"), that is one less way for a malicious entity to gain unauthorized access to a system. It can take as few as one failure in a system's defenses to gain access, so we want to try to eliminate as many of these potential attack vectors as possible.

1.2 Inspiration

While reading a blog post on the spareclockcycles blog [74] about discovering multiple vulnerabilities while searching in the WordPress plugin repository, I began thinking about the massive amounts of untapped data available in various places.

The initial blog entry about WordPress plugins explained that after finding a vulnerability in a WordPress plugin, the author questioned how many other vulnerable WordPress plugins existed. The initial vulnerability discovered was a simple copy and paste from a known-vulnerable Flash gallery library. With a large enough number of WordPress plugins in the plugin repository, the likelihood of some of the plugin authors making simple mistakes that are easy to catch would be pretty high.

By running a few grep [41] operations on the repository, it was trivial to pull out several plugins that had remote file include vulnerabilities. He also turned up some local file include vulnerabilities. After doing some more quick searching, he found a plugin using a vulnerable piece of code (the PHP timthumb image previewing library), and just did some simple searches for inclusion of that code in other projects.

It is interesting how simple all of these vulnerabilities were to find, but taking into account the number of WordPress plugins and the (sometimes dubious) public origin of WordPress plugin code it is not that surprising.

A local checkout of the plugin repository is reportedly around 80 gigabytes in size, and "contains approximately 12,000,000 files." There are actually around 23,000 plugins in the repository, so these numbers are not surprising. He essentially just downloaded the repository and ran grep [41] on many of the files to locate calls and any references to that known-vulnerable timthumb library, as well as a few other obvious security issues.

Of course, these discoveries lead to a discussion on code fingerprinting, and better ways to detect these common issues and defend against the injection and proliferation of such mistakes into various plugins. The conclusion drawn was that this is a huge issue, and needs further exploration and automation. [74]

After reading this post, I came to my own conclusion. This is a very common problem, and it makes sense to be sure to use all the data we have at our disposal. Why not expand the search to include more data?

If it were possible to leverage all the extra metadata found in places such as version control systems and bug trackers, it would be easy to gather a great amount of statistics and do some simple data mining that would help find previously undiscovered vulnerabilities.

In fact, this would require fairly little effort, and once some initial work is done it would be largely automated. It would be feasible to set up an application that essentially monitors your source code or even monitors incoming commits to make sure that there are not any obvious vulnerabilities being introduced.

The way supernothing put it in his blog entry was excellent:

> For the WordPress developers, the best defense would probably be to scan any commits for known vulnerabilities, and either warn or (preferably) block the developers from adding exploitable code to the repository. This can be done quite easily using pre-commit hooks for SVN, which allow for custom verification

> of commits to a repository. I'm planning on releasing an example script when I get time that will detect commits introducing the vulnerabilities I scanned for, but the more interesting problem is how to gather a larger, better collection of signatures. I've got a couple vague ideas for how to go about doing this, but would love suggestions on the subject. [74]

The potential security gains from this approach are hopefully obvious. Catching security vulnerabilities as they are introduced would make it easy to cut down on a large portion of the vulnerabilities found.

Since initially reading the blog entry about discovering vulnerabilities in WordPress plugins, I have given quite a bit of thought to the changes I would make to the existing static code analysis model. Namely, we should be able to augment the set of current tools with additional data gleaned from the available development metadata. In the current static analysis model this development metadata is largely ignored. If we can utilize it to identify potential vulnerabilities, we should be able to make some improvements in the overall security of the tested applications and also potentially in the development lifecycle by providing the developers better feedback.

Chapter 2

Survey of Literature

There are already some existing methods of finding vulnerabilities in software. There is also past research on the collection and cataloging of metadata from various inputs and some interesting research related to identifying similar code using abstract syntax tree comparison. All of these approaches are related to my methods.

2.1 Vulnerability Detection

Static analysis has been practical since the 1970s with pioneers such as Cousot [30] leading the way. Today there are a number of existing static analysis tools in use including some that are free software such as rosecheckers [4], Clang Static Analyzer [6], splint [20], and FindBugs [23]. In addition to many of these more "academic" and open source tools, there are also several proprietary tools available, namely Coverity [8], MAYHEM [27], and Fortify [11]. These tools are already used in the real world on a daily basis to find security issues that need to be addressed.

Most of these tools have a list of tests or checks, similar to plugins, that can be enabled or disabled at runtime. Usually, each test checks for one particular type of known bug, such as the use of an uninitialized variable or a string copy with no bound.

First examine how static analysis tools are commonly used. Understanding a typical vulnerability discovery workflow will help us understand the reason these tools are useful to developers and security professionals.

2.1.1 Workflow

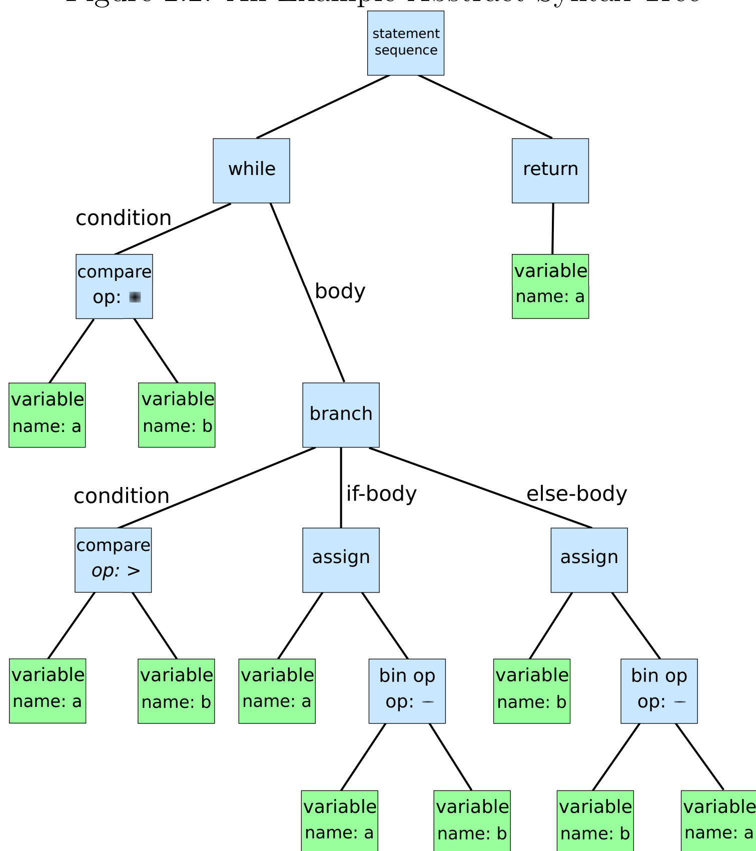
In order for an analyst to perform a white-box application security assessment, he or she will very likely perform some kind of static code analysis. The amount of code to review can be too large to tackle manually, so a static analysis tool is used. It should be notice that empirical studies have indicated that static analysis remains rather academic, and is less often used in practice for finding some types of security issues. [38] It is still relatively easy to discover security vulnerabilities manually, but in the future it is likely that static analysis will prove more and more useful. It is not uncommon for academic research to eventually appear in tools used by the masses.

Most static code analysis tools analyze source code as input, and output a list of areas in the code that should be more closely examined by a human analyst. Some tools, for example ROSE/Compass [18], have the ability to analyze compiled binaries as input, usually flagging points in the code that require further review. The way many of these tools work is by translating either the source or binary code into some kind of intermediate representation, usually similar to an abstract syntax tree (AST). An AST essentially represents the parsed program, where nodes in the AST represent certain syntactic constructs that are semantically relevant. [69] Figure 2.2 displays an AST of the implementation of the Euclidean Algorithm shown in Figure 2.1.

Figure 2.1: Implementation of the Euclidean Algorithm

```
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
return a
```

Figure 2.2: An Example Abstract Syntax Tree



Running the Automated Tool

First, the automated tool is run on the software artifact. Often the easiest way to make sure the tool analyzes the entire application is to integrate it into the build process for the application. Essentially, before each source file is compiled, it is first fed into the static code analyzer.

As the tool works its way through the application, it builds internal representations of the application. These internal representations are often stored in memory as abstract syntax trees (Figure 2.2). From these abstract syntax trees, it is possible to build control-flow graphs representing all the possible paths through the application when it runs. Control-flow graphs are made up of nodes that indicate decision points and edges representing the possible choices for each given decision. Therefore, if there is an if statement/branch, it appears as a split in the graph. [69]

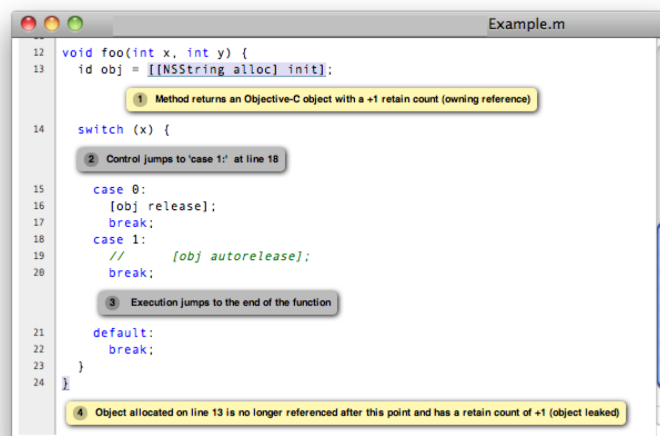
It is possible to use these control-flow graphs to determine what conditions must take place in order for certain code to execute. In other words, it is possible to determine what input data would be required in order to make certain code in the application execute.

The build process for large applications can take a very long time, and with the added complexity of static code analysis it is possible that this step spans multiple days on very large or particularly complex applications. If we can provide similar or supplementary results in less time, developers would be able to find and correct bugs earlier in the development lifecycle.

Analyzing Output from the Tool

As an example of tool output, Figure 2.3 shows some output from the Clang static analyzer [6]. It has located a memory leak in some Objective C code, and demonstrates the conditions that must be fulfilled in order to cause the bug to execute in the application.

Figure 2.3: Clang Static Analyzer



Clang produces an HTML file with annotations added to indicate which conditions lead to the execution of a certain path. The user scrolls through all the files output by the tool to determine the input needed to execute the potentially vulnerable code.

A security analyst examines the output from whatever tool he or she is using, and determines which issues are legitimate issues that require attention, and which are likely either false positives or not a high enough priority to worry about.

Test the Application

As stated above, some issues are deemed low priority, and are safe to ignore. These low-priority issues may be simple things like formatting and coding standard compliance that even if valid do not necessarily indicate a security issue in the application being tested. The remaining issues, however, need to be examined by an analyst.

Code that has been flagged could turn out to be benign, which is referred to as a false positive. Obviously, it is misleading when these false positives are included in a final report so it is important to work to minimize them by testing and verifying.

Testing and debugging are often also useful in determining the exploitability of a given vulnerability. For example, if a vulnerability is confirmed through testing, it may be necessary to use some debugging applications to determine whether an attacker could leverage this vulnerability in an attack vector. Many vulnerabilities do not present a risk for more severe consequences like arbitrary code execution, perhaps allowing for only denial of service conditions or information leakage. [58]

While these vulnerabilities are still potential issues that should be examined and may need to be mitigated, they are arguably less important than a remote code execution vulnerability for obvious reasons. That is not to say that only vulnerabilities that can result in remote code execution are important. It is common for an attacker to use several vulnerabilities to achieve his or her desired effect. For example, achieving remote code execution in a sandboxed application leaves an attacker a limited number of possible post-exploitation activities. He or she would also need to exploit a vulnerability in the sandboxing code itself to "break out" and gain full access to the underlying system. There are many platforms

and applications that use sandboxing, such as Android, and various sandbox-circumventing attacks have been published over the years. [33]

When an attacker is able to execute arbitrary code on a system, it can result in a total system compromise depending on other factors like the privilege level or the level of isolation on the machine. Even if the system administrator is doing everything else right like running the service as a non-privileged user, there will always be other bugs such as privilege escalation flaws that, when used in conjunction with a vulnerability that provides an attacker access, could lead to things like persistence and proliferation, allowing the attacker to pivot off of that machine deeper into the network to which it is connected. [60]

Propose Corrections

The final step in the workflow would be to determine what to do about the bugs found. This is dependent on the goal of the assessment. Assessments can be commissioned for various different reasons. It may be used to demonstrate to management the fact that there are actual recognized flaws in the application in use, to justify a budget increase, or there may be an interest in the remediation of the vulnerabilities found in some manner. [53]

The analyst's job is to figure out the purpose of the assessment and deliver the corresponding results. Often the desired results are in the form of a remediation report, detailing various steps that can be taken to address the issues discovered and sometimes some kind of priority-ordering explaining which issues should be addressed first.

It is very important to be mindful of the client's business needs, and to give some serious thought to which vulnerabilities should really be addressed first. Not every business is the same, and some have higher tolerances for certain types of risks.

2.1.2 Summary

We can see why this static analysis technique is valuable, and hopefully it is clear why identifying and resolving vulnerabilities is desirable. Next we must discuss some of

the existing methods of identifying vulnerabilities in programs. We will primarily focus on static analysis, although the same core ideas in this proposal could be adapted to work with dynamic analysis as well. Both methods identify vulnerabilities, but dynamic analysis is more useful when for whatever reason it is not possible to access the source code of the application being analyzed. Static analysis of source code requires access to that source code, which works well for us since we want to develop a tool that can analyze the history of an entire source code repository.

2.2 Existing Vulnerability Discovery Tools

2.2.1 RASAr

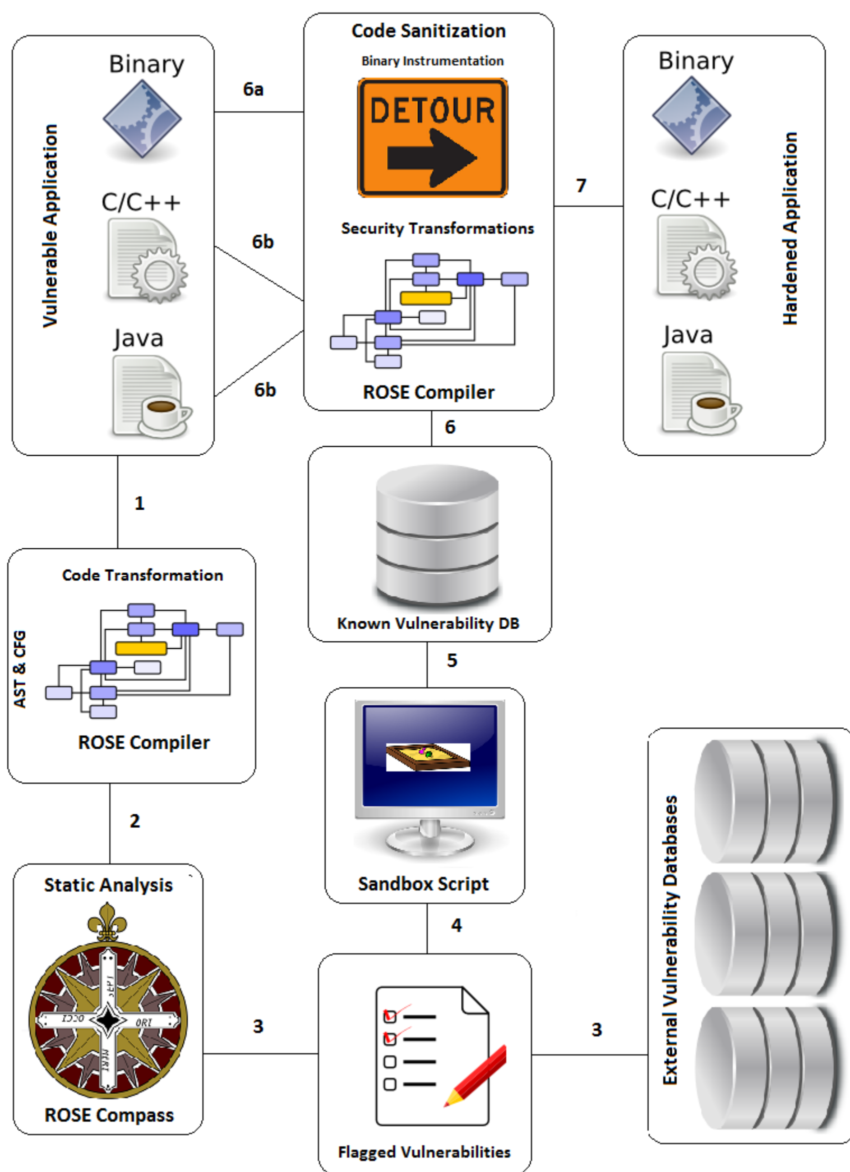
The Information Assurance Lab at Auburn University was researching a method of using a combination of static and dynamic analysis techniques to identify and automatically mitigate vulnerabilities. The tool that implements this method is called RASAr, which stands for Real-time Application Security Analyzer. [48] Steps 1-5 in Figure 2.4 summarize how a standard static analysis tool works.

The primary goal of RASAr is to identify vulnerabilities in legacy binaries and use that knowledge to automatically mitigate the vulnerabilities. The current prototypes of RASAr are built using a combination of other tools, including ROSE/Compass [18], rosecheckers [4], and S2E [28], and leveraging data from external public vulnerability databases such as OSVDB [15].

2.2.2 rosecheckers

The framework we chose to use for RASAr's source code analysis (and some static binary analysis) is called the ROSE Compiler. In order to explain how ROSE can be used to identify vulnerabilities, we look at a project that does just that: rosecheckers [4]. Essentially, rosecheckers is a collection of rules that together enforce the CERT C/C++ secure coding standard [5].

Figure 2.4: RASAr Architecture



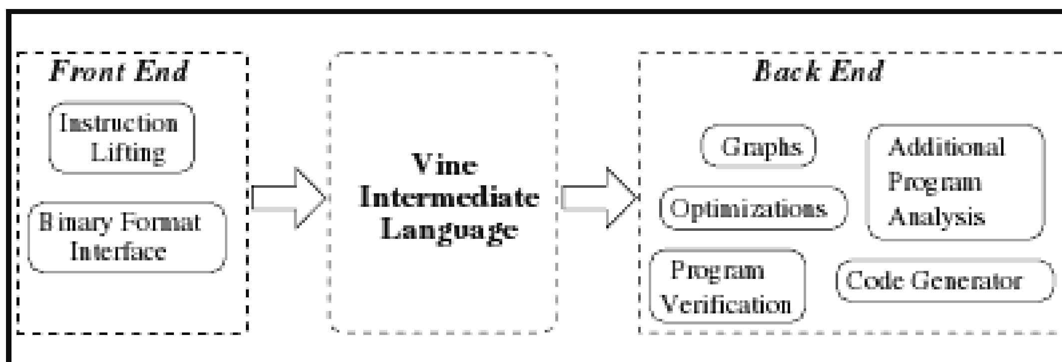
The ROSE Compiler is a framework and API that allows a user to write his or her own code and plugins that interact with ROSE to perform desired transformations, translations, or automated code manipulation. It supports a wide variety of languages as input, and can produce binaries. It works by reading the input file/files in, parsing them, and building an internal representation. This internal representation in ROSE is an abstract syntax tree that can represent constructs found in any program regardless of language. ROSE can even disassemble binary executable files and use them as input, meaning the user can feed it just about any type of program and it can operate on the internal representation in memory. ROSE is flexible, and can be used for a variety of purposes, but the most interesting and relevant to this paper is using it to perform static code analysis.

One of the example utilities written with ROSE is called Compass. Compass is specifically used for performing static analysis to search for bugs or coding style violations, somewhat similar to how lint and its many derivatives [20] work.

2.2.3 BitBlaze

BitBlaze is another framework used for analyzing various types of binaries. BitBlaze uses a combination of static and dynamic analysis when dealing with executable binary code. Among the reasons given, the authors make the point that "static analysis can give more complete results as it covers different execution paths." [73]

Figure 2.5: Vine Architecture Overview



The most relevant part of BitBlaze is the component called “Vine.” As seen in Figure 2.5, Vine also performs the familiar task of parsing the input files (in this case, binary executable files), and creating an internal representation, this time in the form of the “Vine Intermediate Language”. Also, Vine is broken into a back-end and a front-end. Again, however, the intermediate language that Vine uses is not quite as abstract as the abstract syntax trees in use by the ROSE compiler. They are closely tied to the low-level constructs that they represent and are less relevant for higher-level types of input without translating first. The intermediate language used by Vine actually comes from an external language called VEX, which is used in the BitBlaze binary instrumentation tool. This language gets translated to a specific Vine intermediate language. Like the internal representation used by Valgrind [64], the intermediate language used by Vine is closely tied to the x86 instruction set. This makes sense because BitBlaze operates solely on binary executable files, whereas the ROSE compiler has the goal of working with any input file language including both source code and binary executable files.

One interesting note is that Vine’s architecture is modular in a way that makes it possible to use external disassemblers. The authors chose to integrate it with well known disassemblers like IDA Pro [12] as well as their own internal disassembler which is built on top of GNU libopcodes. [10]

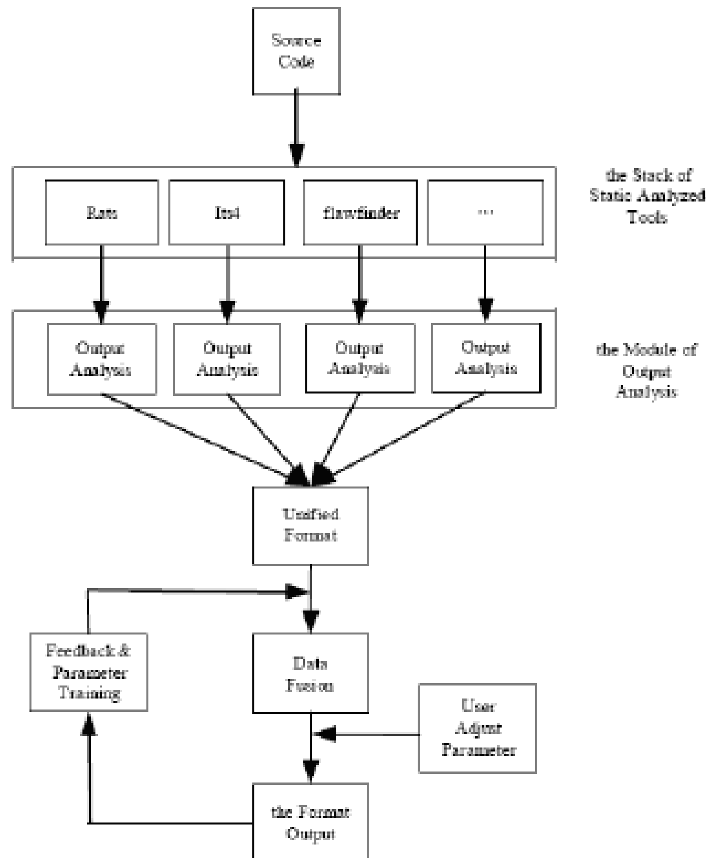
In the Vine backend, there is “an evaluator” (essentially a virtual machine) that can execute code directly from the Vine intermediate representation, allowing for useful features such as state validation.

Interestingly, there are a number of different translations supported using the intermediate representation language in Vine. For instance, it is possible to generate an equivalent C program or even an equivalent program using Dijkstra’s Guarded Command Language. [73]

2.2.4 ISA

ISA [57] is a framework that is very similar to ROSE in that it is used to perform static analysis on source code files and programs.

Figure 2.6: ISA Architecture



What makes ISA interesting is that, in an effort to reduce both false positives and false negatives in static analysis, the researchers use something they called "data fusion," which is using several different tools and comparing the results from each. When used in this manner, multiple flagged issues from the tools can help to confirm each other, thereby reducing false positives. Also, by using a number of different tools, there was a reduced false negative rate as well since each tool uses a different static code analysis technique resulting in complementary output. The researchers looked at the static analysis problem from a

higher-level, and made an attempt at reducing the rates of these two issues, which are both big problems with the current static code analysis utilities available.

The tools used vary widely in the way they go about detecting potential vulnerabilities in the input source code. The researchers looked specifically at tools that use annotation analysis, lexical analysis, grammar analysis, model checking detection, and type analysis. At first glance, this approach may not seem novel or even interesting, but the work the researchers have done in this case is actually in the reconciliation of all the output from the various tools. By combining all the output, they are able to determine which bugs are very likely to be actual vulnerabilities. Furthermore, by using multiple tools they have increased their coverage and decreased the rate of false negatives (vulnerabilities in the source code which are not detected by any tool). The results they show in their paper are clear; their approach successfully decreased the rates of both false positives and false negatives in the final output of the vulnerability report. [57]

2.2.5 MAYHEM

Recent research at Carnegie Mellon University has yielded a static binary analysis tool called MAYHEM. MAYHEM make several novel contributions to the field of static analysis, perhaps the most novel of which is the ability to automatically generate working exploits as proof of the exploitability of the bugs found in the target applications. More relevant to this experiment, however, is the methods MAYHEM uses to identify these vulnerabilities. MAYHEM uses a standard taint analysis and symbolic execution technique, but the authors use some interesting tricks to make the process more efficient.

In order to improve efficiency, MAYHEM does something called hybrid execution, where both symbolic and concrete execution are used to strike a balance between speed and memory usage. A program being analyzed could have many many execution paths, so taint analysis is used in an attempt to focus on the areas where user-controlled input is processed. [27]

2.3 Leveraging Development Metadata

There has been other research done relating to mining development metadata. However, it generally focuses too narrowly on only input, and makes little attempt to leverage multiple sources of information. That means it examines *only* the data in a software repository, or *only* the data in a bugtracker. It is possible to associate vulnerabilities across these boundaries. Doing so gives us a much better picture of why these bugs exist and how we can find others like them. By not looking at all the data we have available to us, we may be missing crucial information that will allow us to make our software more secure.

Researchers Silwerski, Zimmermann, and Zeller were successfully able to identify changesets in version control that induced fixes of bugs. Then, using the change history available from the version control system, they were able to identify changesets that likely introduced the bugs in the first place. [72] It turns out, however, that there are certain common change patterns that occur in the normal development process that make their method less useful. For example, when large changes are introduced that only fix formatting issues (no semantic changes) it shows the committer of that formatting change as the author of that particular line. This causes issues when you attempt to identify bug injection by assuming that the last person to change that particular line has injected the bug. Kim et al have mitigated some of those issues by looking further back in the change history to identify the actual author who introduced the bug in question. They do this by building change trees, which can more comprehensively chronicle the history of a specific line or piece of code. [55]

From these change trees, it is possible to determine when a certain piece of code was added, in addition to the committer that added it. Armed with a more accurate and complete view of the history of a project, it was possible for Kim et al to greatly reduce the number of false positives when attempting to detect the provenance of a bug. [55]

Other research has made attempts at combining information from all available sources [43] [75] [52], but the researchers seemed to stop short of leveraging the data gathered to detect bugs or otherwise find novel correlations. It seems the general trend of research using

FLOSSmole [50] (an open source collection of development metadata for the purposes of data mining) is to study historical data, seeing how certain metrics ended up affecting the trajectory of the project [31] [77] [61], or to study open source project structures [32] and not necessarily to do something like identify previously unknown bugs. I want to focus on using this access to identify security vulnerabilities. Much of this work may be able to be generalized and used to detect other common classes of bugs.

2.4 Manipulating Source Code

There are several different ways for a program to “understand” source code. Our goal is to be able to identify areas of code that are similar to previously identified vulnerable areas. If we simply treat the code as any other text, we can perform standard text matching on it to identify similarities. We can also relax the requirements for matching such that we are searching for partial matches. Finding additional partial matches allows us to identify more sections of code that are similar to our known-vulnerable code, at the cost of a higher false positive rate.

This approach may be sufficient for our purposes, but more sophisticated approaches do exist like AST matching. These more advanced methods actually parse the source code and build an internal model of the syntax or semantics. Having this high-level model of the code allows the user to match code that is syntactically or even semantically similar to previously seen vulnerable code.

2.4.1 Text Matching

Exact text matching is very straightforward. We can use a simple tool such as grep [41] to search for strings in the codebase that are an exact match for signatures. This method will have the most success with finding bugs that are reintroduced verbatim or areas of code that were copied and pasted without modification. Even if the only modifications of the

code are only non-syntax affecting formatting changes, the strings will not match. In order to find code that has been slightly modified, we can perform fuzzy text matching.

Fuzzy text matching, or approximate matching, can be useful for identifying areas of code that are similar, but have been slightly modified. While not a groundbreaking concept, there are multiple different algorithms for performing approximate matching. Ukkonen explains one such algorithm, which uses n-grams (or "q-grams" in the article) to find the maximal match based on edit distance. This approach is fast and efficient. [76]

There are a number of command-line tools for performing approximate text matching [21] including agrep [78]. Agrep differs from the regular grep utility in several ways, including being record-based instead of simply line-based. By operating on records, it is actually possible to do things like extract procedures from source code files. [78] Since agrep is mostly compatible with grep's syntax, it is usually possible to use agrep as a drop-in replacement for grep.

2.4.2 Token Matching

Research from Kamiya et al details a manner of performing code clone detection. Their method, named CCFinder [54], actually tokenizes the input data before performing the clone detection. Doing so allows them to easily ignore changes that do not affect either the syntax or semantics of the code. The authors outline the following goals of their approach:

- The tool should be industrial strength, and be applicable to a million-line size system within affordable computation time and memory usage.
- A clone detection system should have ability to select clones or to report only helpful information for user to examine clones, since large number of clones is expected to be found in large software systems. In other words, the code portions, such as short ones inside single lines and sequence of numbers for table initialization, may be clones, but they would not be useful for the

users. A clone detection system that removes such clones with heuristic knowledge improves effectiveness of clone analysis process.

- Renaming variables or editing pasted code after copy-and-paste makes a slightly different pair of code portions. These code portions have to be effectively detected.
- The language dependent parts of the tool should be limited to a small size, and the tool has to be easily adaptable to many other languages.

[54]

The authors create a set of rules used to transform and normalize the input code so that it can be compared with other code snippets in order to detect clones. These rules include removing namespace attribution, removing template parameters, removing initialization lists, and removing keywords that do not affect semantics. [54]

Ducasse et al have also performed language independent clone detection, using similar methods that predate CCFinder by more than a decade. The authors utilized a similar approach, parsing the input data and then performing some transformations on it to create a tokenized intermediate representation that can be used to compare to other sections of code in order to identify clones. Their approach, however, was less thorough in comparison to CCFinder. The authors use algorithms similar to those employed by the unix diff utility [42], and do not perform as many simplifying transformations on the code in order to normalize it. [35]

2.4.3 Matching Abstract Syntax Trees

Research has shown that it is possible to use abstract syntax trees for matching similar or identical code. Detecting semantically equivalent programs is undecidable, so approximate matching is necessary. Several strategies are employed to be able to find both exact and

“near-miss” clones. Using abstract syntax trees is still more fruitful than using a string-based matching method. [24]

Preprocessors and preprocessing directives can serve to complicate AST generation, but there have been methods proposed to incorporate these preprocessing directives directly into the AST. Doing so would allow us to treat the resulting AST like any other, and would allow us to compare ASTs that have been derived from code with preprocessing directives as well as code that was unencumbered by preprocessing directives. [65] For our purposes, one-way AST generation is sufficient. We do not intend to convert the resulting ASTs back into source code, so as long as they preserve the general semantics of the original source code we should be able to use them to detect clones of vulnerable code. It is of course necessary to accurately preserve location information, so that any identified clones may be traced back to a location in source that must be reviewed by a security analyst and patched in order to resolve any vulnerability identified.

Another research project has used AST matching to analyze differences between consecutive versions of code in a source code repository. This research is doubly relevant to us because it not only uses AST matching, but also performs that matching on code pulled directly from version control. Of course the authors ignore all of the metadata in the repository, but they are at least working in a similar direction. [63]

The primary interest of these authors was researching the ability to do dynamic software updating (modification of live code), and, as it turns out, knowing how the software changes between updates is important. In the future it should be possible to apply techniques from both of these projects in order to find new vulnerabilities that are syntactically and semantically similar to known vulnerabilities.

2.4.4 Refactoring Tools

One class of tools that necessitates the programmatic comprehension of source code is refactoring tools. A refactoring tool makes it possible for a programmer to perform behavior-preserving transformations on source code in an automated fashion. For example, a refactoring tool may allow a programmer to change all occurrences of a variable or function name or even make large-scale changes to the architecture of a program while preserving the semantics. Sometimes syntactic changes are also made to a language and old code must be updated. Refactoring tools can make this process much easier. [66] [68]

Dr. Jeffrey Overbey, in his Ph.D. thesis, proposes a toolkit for constructing refactoring engines. His work highlights some of the difficulties in programmatically manipulating source code. The portion of a refactoring tool that performs the actual AST transformations can be an order of magnitude smaller (as measured by lines of code) than the portion of the tool that converts the actual source code into an intermediate representation/AST that can be easily manipulated. This larger portion of code is referred to as "refactoring infrastructure", and it is from this infrastructure that we can glean useful strategies for detecting vulnerable code in an application. [66]

While we do not require a rewritable AST for performing clone detection, Dr. Overbey shows that it is possible to create a parser and AST generator automatically from a language's grammar specification. Similarly to the requirements for an AST in a refactoring tool, we would need an AST to provide a precise source-location mapping. However, our other requirements are more relaxed. We would not necessarily require the AST to maintain an accurate model of the original source code, because all we are really interested in discovering is whether or not pieces of code are semantically equivalent. [66] [67]

If we wanted to increase the number of semantic clones detected, we could perform semantics-preserving transformations on a rewritable AST in order to enumerate potential clone candidates.

Many refactoring tools exist as plugins to integrated development environments like Eclipse and NetBeans. Refactoring tools have come to be expected, and are a useful feature to have when working on the development of a large software project. The tools have come a long way, but still have their issues. Some of these issues include dealing with code containing preprocessor directives and issues with certain common refactoring patterns like EXTRACT INTERFACE (which is used to extract an inheritable interface given a certain subset of members). [65] [40]

Dr. Munawar Hafiz proposes a catalog of many different refactorings/transformations intended to increase the security of an application. These refactorings cover a wide variety of security issues, which in aggregate represent a rather comprehensive collection. Most of the transformations described are abstract, and could theoretically be applied either directly to source or to a rewritable AST. [45] Some transformations can occur at another layer entirely, like those proposed by Hafiz et al for preventing injection attacks. [46]

With any luck, perhaps one day automated vulnerability detection will come as a standard plugin to popular integrated development environments. Many of them already integrate with common revision control systems and bug trackers.

Chapter 3

Experimental Design

In addition to looking for bugs in software, we should be looking for trends.

Static analysis has been finding bugs for quite some time, but there is a wealth of additional information available aside from just the latest version of the source code or binary. In fact, there is so much untapped data that it may seem like a daunting task to analyze it. There exist techniques for harnessing previously ignored data to elucidate patterns in a way that is helpful to developers. These techniques are known as data mining, and it is a very active and expansive field in computer science. I show that by applying some simple data mining techniques we can achieve some quick wins that in turn find us some new and valuable data.

It would be unheard of for a contemporary development team not to use some kind of version control system for their code. Some of the first of these included CVS [7] and Subversion [3], but today there are some great distributed version control systems such as Mercurial [14] and Git [9]. With these systems, the entire history of the project is available to users locally. Why ignore all revisions but the latest when analyzing the code? By looking back at the history, we can see things such as the committers of code to the project, and when they committed it. We can see commits that are a direct result of fixing particular bugs. We can gather useful and interesting statistics about commit ratios, defect injection, etc.

Of course, there is even more information we can get our hands on to aid in our analysis. For example, most development teams use some kind of bug tracking/ticketing system to prioritize features, report bugs and vulnerabilities, and discuss various design issues. Often these systems integrate somehow with the development team's version control system. We

can leverage this information to gain even more insight into vulnerabilities in the application. When a bug is reported, details are posted to that ticket, the status of the ticket is updated, patches are posted on the ticket, and the ticket is eventually closed. There is a great wealth of metadata here that can be used to identify additional bugs.

At a high level, I have implemented a familiar approach [1] [57], which looks at a number of different triggers or signals in order to compute the relative likelihood of certain vulnerabilities in an application. Using this approach, it is possible to find more issues in the code in a manner that produces a manageable number of false positives.

The central hypothesis in this dissertation is that for a given a project, we should be able to use the metadata created from its development to identify additional, previously unknown, vulnerabilities. I have designed an experiment that will prove this hypothesis via demonstration, using code and metadata from a couple of popular open source projects as data sets. As a corollary, we will show that metadata from one project can be useful for finding vulnerabilities in another project. Finally, we will show that as a byproduct of this approach, we can generate useful development metrics that could aid further vulnerability discovery and could be used to identify potential problem areas in the development of a project.

The difference between existing tools and my work is like the difference between trying to get to know a person by looking at their life as it currently is, and trying to get to know them by learning about the changes they have experienced throughout their entire life.

Existing tools look at only one version of an application's source code. The new approach that I propose involves looking at the entire history of the project (as well as the histories from other projects), and using past mistakes to detect and fix new mistakes. If we can learn to detect risky development patterns and vulnerable code signatures, we should be able to automatically identify new vulnerabilities and possibly even prevent developers from committing vulnerable code in the future.

I have been able to demonstrate this concept using two test cases: Python (specifically cpython) and Roundup. By analyzing the metadata, I show that it is possible to detect new bugs in the latest version of the source code that have not been previously reported. The bugs found are semantically similar to previously resolved vulnerabilities.

3.1 Architecture

In order to find new vulnerabilities, and thus prove my thesis, we need to build a tool that can take into account all of the available development metadata and leverage it to detect similar vulnerabilities in the current version of the source code for the project. We will accomplish this by using a combination of data aggregation, machine learning, and code clone detection techniques.

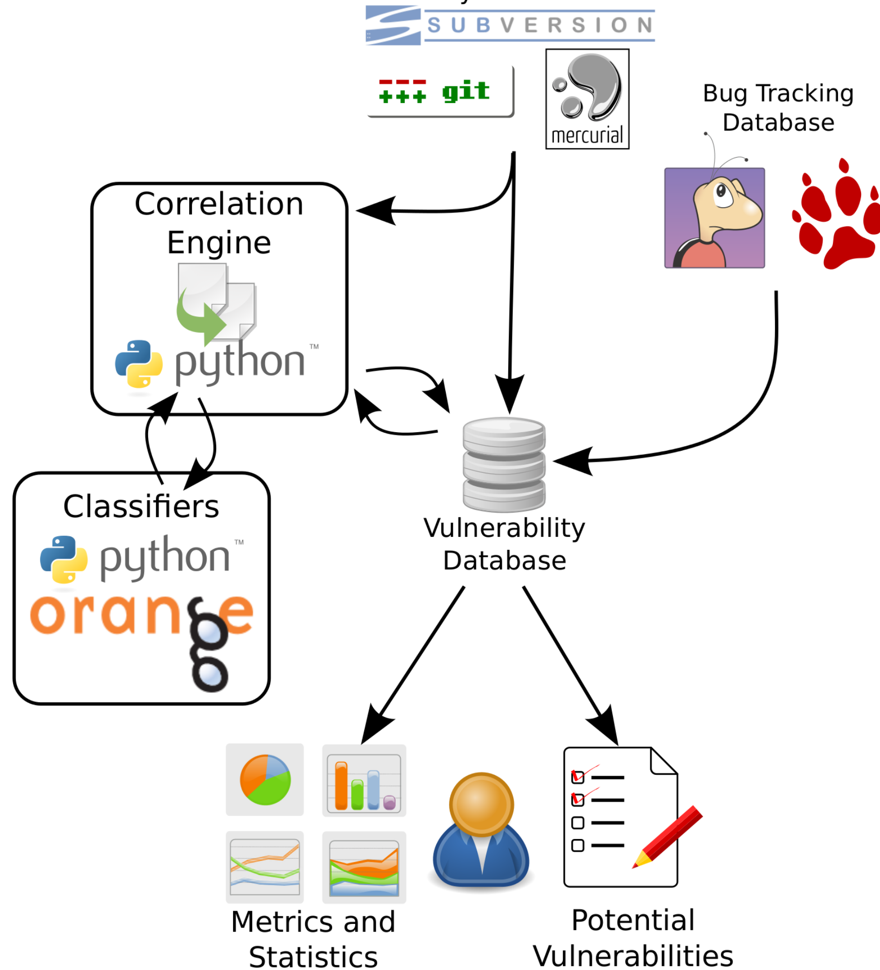
By taking all of this extra data into account, the architecture of this new tool is significantly different (Figure 3.1) than a traditional static analysis tool. Instead of solely relying on the most recent version of the source code as input, the tool takes into account the history of all versions in the source code repository.

The benefits of using a distributed version control system become apparent here. With distributed version control systems, a copy of the entire repository is stored locally, so it is easy and very fast to switch the working directory to an old historical revision of the source code.

If all that is available is a non-distributed version control system, such as CVS or Subversion, then it is advised to make a full local copy of the repository. Otherwise, checking different revisions in the repository history will result in superfluous network traffic and the potentially high latency associated with it.

The same goes for the bug tracker. Having a local copy of the bug tracker will make the analysis much faster, as there will not be a need to (potentially repeatedly) query a remote web server for every bit of information needed.

Figure 3.1: Project Architecture



With all this data available, it is often possible to go directly to the source code and find the actual commit that resolved the vulnerability. Then, by searching the history of that file we can often determine the changeset that introduced the vulnerability itself. Finally, from the source code, it is sometimes possible to identify part of the vulnerable code that can be used to search for similar vulnerabilities in the current version of the project.

Not only could a user find vulnerabilities elsewhere in the codebase, but also in other codebases entirely. In other words, searching through other versions or even different applications can potentially yield even more matches, and thus more vulnerabilities discovered.

The static analysis and data mining tools will produce a list of potential vulnerabilities as well as useful metrics and statistics over the data found in the repositories and bug trackers.

These metrics include data such as committers with high vulnerability injection rates, locations of code that frequently contain vulnerabilities, and types of vulnerabilities that were easy or difficult to find judging by comparing the time they were injected with the time that they were fixed.

All of this data is available and accessible by combining information from the bug tracker and version control system, but until now nobody has really tried to correlate the two and leverage the wealth of development metadata that exists.

The vulnerability data gathered can be stored in a local vulnerability database. More importantly, though, the database can link all relevant vulnerability data. Each vulnerability may have associated with it multiple bug reports, multiple commits (introduction and resolution), multiple locations in the code (and therefore multiple abstract syntax tree representations).

So, from the users' perspective, they feed some source code in as input. This input source code can be from a current project or perhaps a new commit as discussed above. The static analysis tools parse and inspect the input files and compare them with known-vulnerable code samples stored in the vulnerability database and outputs a list of potential vulnerabilities.

This experiment adds to some existing research that takes advantage of having a full history of the codebase in a version control system. When finding new bugs I use two approaches: a simple text matching approach, and a machine learning approach. The text matching approach tries to identify known bugs in the current codebase. The machine learning approach trains a classifier using the labeled commits and uses it to predict the classification of other commits.

In order to show that it is possible to derive useful information in the form of previously unknown vulnerabilities from development metadata, I propose an experiment. The general methodology for this experiment is as follows:

1. Data gathering and organization
2. Bug correlation
    - Search issues for commit references
    - Search commits for issue references
3. Bug injection identification
    - Ignore documentation or test changes
    - Find injection using VCS annotation
    - *Find injection using added test cases and VCS bisection*
4. *Find new bugs*
    - *Identify similar code via text matching*
    - *Identify similar commits via machine learning*
        - *Gather statistics for all commits*
        - *Train on commits that have injected bugs*
        - *Find similar commits*
5. Manual Verification

The emphasized lines indicate the improvements and contributions made by this research. These steps are novel, and I will show that they can be used to identify previously unknown vulnerabilities in the current code base for an application.

In other words, this method of identifying vulnerabilities in source code is different from the current methods that are typically used. While static analysis of source code is common,

this approach takes advantage of previously identified vulnerabilities to locate other similar areas of code that may also be vulnerable. If successful, this new approach can be used as another tool that developers can take advantage of in order to ensure the code they are writing is secure.

3.2 Data Gathering and Organization

First we need to gather all the relevant data that will be used to find vulnerabilities. Initially, the data sources considered included the source code repository, bugs from the bug tracker that have been marked as security issues, the respective messages from the bug tracker concerning those issues, and all e-mail messages from the public mailing lists. With this data, it is possible to correlate security issues with any related changesets or messages.

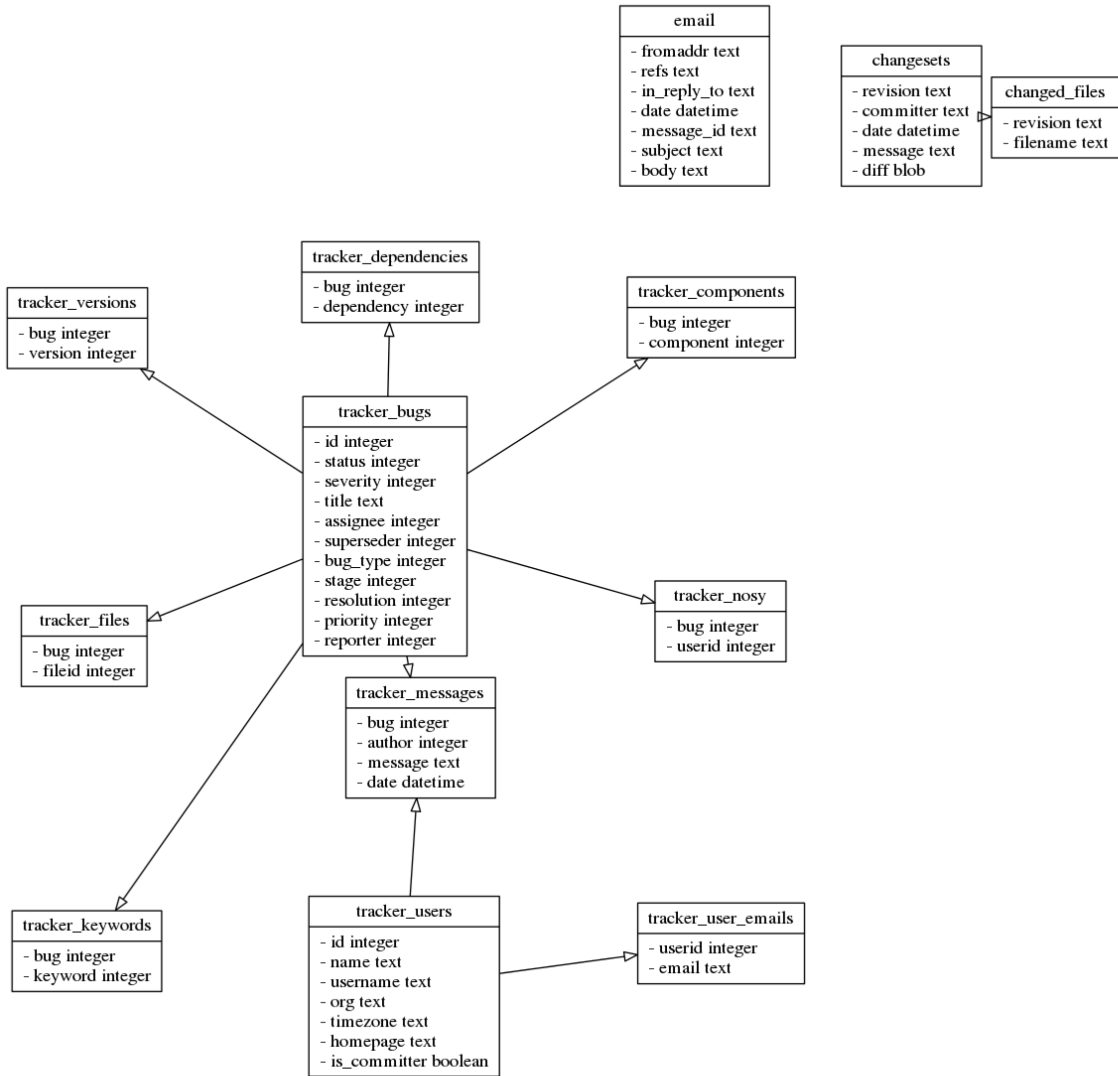
However, it turns out that having the e-mail messages from the public mailing lists is not as beneficial as originally theorized. It is significantly more useful to have tagged issues in the bug tracker so they can be filtered to contain only the types of bugs that we are interested in locating.

Obviously by making local copies of all the input data, we are able to speed up the analysis process significantly. This local cache can be stored in a relational database to make it easy to query. Figure 3.2 shows the schema of the local cache/DB.

3.3 Bug Correlation

With all the data collected, the next step is to correlate all of the bugs. From the tagged bug tracker data, we can create a list of vulnerabilities that have been found and reported. There is a wealth of useful data in the tracker entries, but in order to perform the bug correlation we only need to scan the messages attached to each bug entry. This process allows us to establish links between the bug reports and related changesets in the version control system.

Figure 3.2: Local Cache/DB Schema



We can identify changeset mentions by searching for actual hyperlinks to specific changesets or by searching for mentions of specific changeset IDs. For distributed version control systems like git and Mercurial, changeset IDs are usually specified as a hash, and it's possible to reference them using an unambiguous prefix. Commonly, however, a short 12-character hex prefix is used to reference an individual changeset.

These mentions may be indicating a changeset that fixed the bug, a changeset that introduced the bug, or just a changeset with some relevant information to the discussing taking place in the messages attached to that particular issue on the bug tracker. Without

further analysis it is relatively difficult to determine why the changeset was mentioned. Still, it is somehow related to this bug, so it could be useful for finding similar commits that indicate other bugs so we add it to this bug's list of related changesets.

Next we scan through the cached data for each changeset in the version control system in order to find links back to issues in the bug tracker. These may be mentions of certain issues by ID, or may be an actual hyperlink to a specific issue in the bug tracker. We can reasonably assume that these changesets are resolving the issue mentioned (as determined by manual examination). Even if some of our assumptions turn out to be incorrect, this anomaly will only lead to a slightly increased false positive rate. It is better to err on the side of caution and include these changesets that we are less than 100% confident about.

Sometimes it is possible to determine with certainty that a certain changeset resolves a specific issue. Often commit messages in a changeset will explicitly state that they resolve a specific issue, or there may be a message posted to the issue in the bugtracker stating that a certain changeset resolves the issue (and the issue is usually also then marked as "closed").

For the Python codebase, this automated process identifies 305 bug resolution changesets from 76 security issues. It may require multiple changesets in order to consider the issue closed, because changes often have to be merged across different branches in the source code repository. In other words, many of these changesets will be similar.

3.4 Bug Injection Identification

In addition to these "related changesets" and "resolving changesets" for each bug, we want to maintain a list of changesets that we believe to have injected the bug. Identifying these changesets that have injected a particular bug is accomplished via two methods. The first method is an improvement on the SZZ algorithm [72] as outlined by Kim et al. [55] The second method is a novel approach that leverages a common feature in version control systems called bisection in an effort to catch bug injection that current methods may miss.

3.4.1 Find Injection Using VCS Annotation

Version control systems provide a great benefit to development by allowing users to track the history of a project. History tracking works by keeping track of all changes made to the code, plus some additional metadata. When a developer commits a change to their repository, that change is logged along with information like the username of the developer making the change, the time the change was made, and a comment from the developer about the change. Therefore, we are able to look up the author of every line in every file that is in the repository. Additionally, when a line is changed, we are able to go back and look up the last user to edit that particular line. This feature is sometimes called “blame” or “praise”, but more generally it is known as “VCS Annotation”.

Once we have identified a changeset that resolves an issue, we can use this annotation feature to find the last user who touched that particular line or area of code. This method is very straightforward and does work reasonably well, but as Kim et al noted in their research, it is too simplistic. [55] For example, formatting changes (or any change that does not have an impact on the semantics of the code in question, for that matter) will change the annotation such that the user making the formatting change is listed as the last author of that particular line of code. In other words, if user A originally introduced a bug on line 42 of a C program, but user B later goes through the same file and changes the indentation, then all the lines will now show user B as the author. This is problematic because now it looks like user B has introduced the bug, when actually the bug had already existed.

Kim et al explain that it is possible to construct a tree of the history for a specific change, allowing us to track the line back to its original author, and likely the true point of injection for that particular bug. They call this tree an annotation graph, and find that using it reduces false positives by up to 2%, and false negatives by up to 4%. [55]

Next they exclude changes to blank lines and comments, which seems like a bit of an oversight on the part of the creators of the SZZ algorithm. Nevertheless, this exclusion results in a reported reduction in false positives by up to 20%. [55]

They also make an attempt to exclude format changes, which can include adjustments in indentation or purely stylistic changes. By ignoring these format changes, the authors report reductions in false positives by up to 25%, and reductions in false negatives by up to 14%. [55] They are a bit unclear about how they determine whether or not a change is merely a format change, once it is determined that a change is a format change it is possible to use the annotation graph to find the original author of that piece of code.

The last improvement they make is to reduce the number of changesets they are looking at by excluding large "bugfix" changesets, where multiple issues are all fixed at once. Sometimes these can indicate that a branch in the version control system is being merged with another branch. Merges do not usually include many new changes; they are usually just a method of applying previous changes to a separate branch of development. They consider all large changesets to be outliers, and subsequently exclude them from the analysis. Doing so further reduces the false positive rate by up to 16%. [55]

My methods closely mirror those outlined by Kim et al. The detection of format changes is relatively basic, but I am somewhat less interested in the reduction of false positives than I am in the reduction of false negatives. It does not matter much as long as the number of potential new vulnerabilities identified is small enough for one person to go through and manually audit in a reasonable amount of time.

For the Python codebase, this automated process yields an identification of 134 bug injections from the 305 bug resolutions previously identified.

3.4.2 Find injection using added test cases and VCS bisection

The Python project, as well as several other open source projects, require that in order for an issue to be considered "resolved" the patch must include a unit test that demonstrates the error. This requirement is not always enforced, but it is sometimes possible to

leverage these added test cases to improve our bug injection determinations. Unit tests represent a common black-box testing strategy that depend on program behavior and functional requirements, and can be useful for testing for security vulnerabilities. [62]

We can take advantage of these test cases by first identifying that a test case has been added, and then running the new test case against different versions of the software from the history of the project.

Version control system bisection provides a very useful tool for this task. For example, if an issue is resolved and a test case is added in revision 142, we know that the bug must have been injected before that revision. Performing a bisection allows us to quickly determine where the bug was likely injected by eliminating many revisions as potential injection candidates at once. We can go back to revision 75 and test the software to determine whether or not the bug is present. If it is indeed present, the test case fails and the bug must have been injected before (or by) that revision. If the test case passes, we know that the bug does not exist in this revision and must have been injected before this revision. This search is similar to a binary search, and is `O(log n)` in time complexity with respect to the number of revisions in the source code repository.

Still, this method can be rather time consuming. For each changeset marked as resolving an issue that also adds a test case, we have to perform a bisection on the repository. Bisection involves updating the working directory to a certain specific revision in the history of the project, then compiling that version of the project, and finally running the test case to determine whether the test case passes or fails. This process can be sped up somewhat by testing for multiple issues at each test revision.

In the bug set for Python, there are 120 changesets marked as resolving an issue that also add a test case. It takes approximately five hours to use this method to identify the changeset candidates that have likely injected the bugs.

3.5 Find New Bugs

The main goal of this experiment was to show that we can use the knowledge of previous bugs to identify new vulnerabilities in software. The methods outlined in this dissertation are meant to be use in addition to traditional static analysis tools, and do a good job of catching issues that are similar to other issues that have been previously identified in a project.

Thus far, we have taken data from various sources commonly used in the development process of a project and identified a list of changesets that likely injected vulnerabilities into the codebase. What can be done to utilize this information?

3.5.1 Text Matching

One way we can leverage these issues is by using them to search for similar bugs. If we have identified areas of code that introduce vulnerabilities, then we can search through the latest version of code in the repository for similar code.

The easiest way to scan the codebase for vulnerable code is to identify unique snippets of code from the bug injecting changesets and perform a simple grep [41] for those snippets. It is best if the snippets are not common statements, so it makes sense to ignore pieces of code that will result in a deluge of false positives.

The changesets that have been identified as resolving vulnerabilities have enough information in them to identify vulnerable lines without even looking at the changesets we have identified as injecting the vulnerabilities. A changeset representing the resolution of a vulnerability will have the vulnerable lines marked as removed in the diff. A changeset is essentially just a diff/patch with some metadata attached.

Obviously lines that are comments or empty should be ignored. It also makes sense to exclude large changesets (outliers, or bugfix changesets) as noted by Kim et al. [55] What remains is targeted fixes that change or remove only a few lines, which can be used to search for similar code in the repository. Some changesets that resolve vulnerabilities may only add code. It may be necessary to check certain conditions before executing certain areas of

code (for instance, to check whether a user is authenticated or has the required permissions to perform that action). In these cases it is more difficult to determine how to find similar vulnerabilities via text matching. It is still possible, however, to attempt to determine when that area of code was last modified and use the machine learning approach to try to identify similar issues.

The text matching method is most useful for catching two types of bug injection: vulnerable code reuse and vulnerability reintroduction (regression). The latter illustrates the importance of requiring test cases for all changes to the code. If a unit test is created to test for the existence of a vulnerability, then that test case will fail if the vulnerability is later reintroduced. Vulnerable code reuse occurs when vulnerable code is copied and pasted to a different location. If a bug is found and fixed in the first version, the committer fixing the bug may not know that the same vulnerability exists elsewhere in the codebase. During testing of the Python project, both types of bug injection were identified using this method.

3.5.2 Machine Learning

The other way that we can utilize the data to discover new vulnerabilities is by applying machine learning techniques to find patterns and identify some indicators of the likelihood that a changeset introduces a vulnerability.

In this experiment, we utilized Orange [34], which is a data mining framework written in C++ and Python. Orange comes with several different machine learning algorithms that can be used to train a classifier based on a labelled input data set. This classifier can then be used to classify new or previously unseen data.

There are a number of possible signals that could be used to indicate the existence of a vulnerability in an application. Each of these signals give us additional insight into the development process, and can pinpoint locations in the code that have higher likelihoods of containing vulnerabilities.

In order to identify which signals are useful for differentiating changesets that introduce vulnerabilities from all the other changesets, we can analyze the information gain for each feature we have available to us.

Information gain is a way of quantifying the usefulness of certain features when looking to differentiate various objects for the purpose of classification. For example, a feature will have low information gain if there is little variance between objects in the training data set. [70]

There are many different machine learning algorithms that exist for performing classification, and it is actually possible to combine them via a technique called ensemble stacking. [36] This technique can often show an improvement over using an individual classifier, because it tends to mitigate some of the weaknesses of individual classifiers. [47]

The main machine learning algorithms that are used in this experiment are C4.5 [71] and Association Rule Mining [22]. The C4.5 algorithm can produce decision trees for classification. Association Rule Mining can be used to find patterns in the features that occur in similarly classified objects in a data set. These rules can then be used to classify new objects. [59]

The signals we examine to identify potential vulnerabilities include the following:

Committer's Bug Injection Rate

Inevitably, certain committers will commit more bugs than others. By searching the repository, it is possible to calculate how many bugs each committer injects per line of code they commit. Committers with higher bug injection rates should be watched more closely, and their commits closely scrutinized.

Appearance of Known-Vulnerable Code

Similarly, any bug that has been fixed in the past can be added to the list of "known-vulnerable code". The repository can be searched for inclusion of such known-vulnerable

code, including code from other projects. There are often common libraries that are included without modification in a large number of different projects. If one of these libraries (or perhaps an old version of the library) is found to be vulnerable, then its appearance would imply a vulnerability in the code.

The use of the PHP timthumb preview code as mentioned previously on page 3 is a great example of the usefulness of this signal.

Historically Buggy Code

For each bug identified, there is some useful metadata associated in the form of project location. If one particular function/class/file has a long history of vulnerabilities that have been discovered in it, that may indicate that there are still some left to be found. Alternatively, changes to this area of code may need to be more carefully scrutinized.

This metric can also be useful for identifying potential sections in a codebase that would make good candidates for refactoring. If code has been historically buggy, maybe there is an underlying reason. Maybe the security model needs to be reevaluated for this code. Maybe it is too complex to maintain without accidentally introducing vulnerabilities.

Commit Time

Many programmers are more productive at night, but perhaps that impacts the quality of code being submitted. Is code submitted at certain times of the day or night (or even month or year) more likely to contain vulnerabilities?

3.6 Manual Verification

Once we have our list of potential vulnerabilities flagged in the current version of the project's source code, we must go through the results and manually verify that the findings are true positives. Typically this requires a moderate level of understanding of the application's architecture as well as a knowledge of vulnerable design patterns. Certain code

idioms may be perfectly safe under the correct circumstances, but can introduce vulnerabilities when used incorrectly. For example, if data controlled by the user is ever evaluated as code it can introduce a command injection vulnerability. We often see these types of vulnerabilities in web applications, and they take the form of SQL injection, shell command injection, or cross-site scripting.

In order to manually verify any potential vulnerabilities that are flagged using the previously described methods, we must examine the context of the identified source code. If it is not possible to exploit the code, then it is considered a false positive. The exploitability can often be determined by manual inspection (static) or manual testing (dynamic). After pruning the list down to only the true positives, we have our final results.

3.7 Resolve Vulnerabilities

One benefit of detecting new vulnerabilities that are similar to previously fixed vulnerabilities is that we can often reuse the same fix for the new vulnerability with little to no modification. While the tools created for this dissertation to demonstrate these detection methods do not automatically provide resolutions in the form of a patch, it should sometimes be possible to do so. [45] We already know the lines of code that match the known-vulnerable code signature, and we know what fixes were applied to resolve that past vulnerability. Therefore, it should be possible to automatically provide a patch that reproduces the modifications made to fix the similar vulnerability previously.

Additionally, when reporting bugs found in a project that are similar to other bugs, we can use past discussion and fixes to aid our resolution of the new vulnerabilities. Doing so, along with providing a useful patch that includes the fix and unit tests, can greatly help reduce the amount of time it takes for the core developers on the project to accept the changes.

Chapter 4

Results and Validation

The validation of the results from this experiment is fairly straightforward. The initial goal of finding previously undiscovered vulnerabilities has obviously been met. Judging how well this approach works compared to existing tools is a bit more complicated, considering the fact that there are no other tools that work in this same manner. Additionally, this method is not meant as a replacement to traditional static (or dynamic) analysis tools. This method is meant to be used to supplement the findings from existing tools.

In summary, there are three criteria for validation:

- Does it find 0days?
- Does it have a reasonable amount of false positives?
- Does it have a reasonable amount of false negatives?

The first two criteria are the most important. Obviously if the methods were not capable of finding new vulnerabilities, the methods would not provide any additional benefits and would therefore be entirely useless.

The second criteria is interesting. How do we determine what a reasonable amount of false positives may be? This number may not always be the same. For example, on a very large development team it may be acceptable to have many false positives because you have many quality assurance developers who are devoted to analyzing potential vulnerabilities. At the same time, though, you still want to minimize the number of these issues because it is possible to become desensitized when facing an unreasonable number of false positives. It also depends on how these issues are presented to the developers.

Conservatively, an analyst needs to be able to process the entirety of the results in one sitting in a reasonable amount of time. If it takes more than about thirty minutes to sift through all the false positives in your results, then it becomes a larger task that must be integrated into the overall build/release process.

False positives can be verified manually, but false negatives are usually more difficult to detect. While it is always desirable to find as many new vulnerabilities as possible, the methods outlined in this dissertation are best at identifying vulnerabilities that are similar to existing issues that have been reported and fixed. Even though vulnerabilities are often similar, unfortunately not all of them are. There are always new types of vulnerabilities discovered, and always new code discovered to be vulnerable that may not be vulnerable in other contexts within an application.

4.1 Python Results

The first data set that this experiment used for evaluation was the Python project, or more precisely, cpython. The cpython codebase is the reference implementation for the Python programming language. The interpreter itself is written mostly in C, and the project includes a large standard library (stdlib) that is a mix of Python modules written in both Python and C. The Python developers switched to a distributed version control system (Mercurial [14]) in 2009 [2] and have used a version of Roundup as the public bug tracking system that currently contains over 34,000 issues. [16]

The source code repository was initialized in 1990 (as a subversion repository) and now contains over 93,000 commits by at least 192 different committers. There are 25 commits in the repository that do not have a committer listed. The codebase is nearly 1 million lines of code as counted by Open Hub (formerly Ohloh). [17]

The Python project makes an excellent test case because of the large codebase utilizing multiple programming languages, long development history, healthy development practices, and high availability of development metadata. The core developers enforce a policy that

requires all changes to code must come with updated unit tests and documentation. It is this unit test requirement that makes it possible to use unit tests to determine when vulnerabilities were injected. It is also useful that on the Python bug tracker security issues are already labeled separately from the other issues. Unfortunately, some projects like the Linux kernel [13] do not separately label security issues. If that happens to be the case, we would need to manually label the security issues, which may be time consuming.

I am already quite familiar with the development community and the cpython codebase (both the interpreter and stdlib), which makes it easy to analyze the list of flagged vulnerabilities and determine which are true positives. I was very quickly able to find two issues using only the text matching method.

- http://bugs.python.org/issue22419
- http://bugs.python.org/issue22421

The output from the tool that led to the discovery of these vulnerabilities is as follows:

```
[*] issue10714{closed|security|msg:4|related:3}
[ ] self.raw_requestline = self.rfile.readline()
cpython/Lib/wsgiref/simple_server.py: self.raw_requestline = self.rfile.readline()

[*] issue672656{closed|security|msg:5|related:2}
[ ] self.address = ('', port)
cpython/Lib/pydoc.py: self.address = ('', port)
```

The first issue was a denial of service issue identified in the reference WSGI (Web Server Gateway Interface) implementation. There was a snippet of vulnerable code reused from the implementation of the HTTP server provided in the standard library. A malicious client could send a very long response that would cause the server to hang. The bug was fixed in the HTTP server, but was not fixed in the WSGI implementation.

The second issue was an information disclosure issue in the pydoc server, which is used to provide local access to Python's built-in reference documentation via a web browser.

Developer	Rate
Dev00	14.29%
Dev01	4.35%
Dev02	3.23%
Dev03	2.78%
Dev04	2.53%
Dev05	2.08%
Dev06	1.59%
Dev07	1.59%
Dev08	1.19%
Dev09	1.15%

Table 4.1: Python Developer Vulnerability Injection Rate (Top 10)

The server was configured to listen on 0.0.0.0 instead of only listening on the loopback address, thereby making the accessible to anyone on the same network. On its own, this bug potentially discloses information to attackers like the version of Python installed on the host. At first this may not seem like an important issue, but it could potentially be coupled with another unknown vulnerability in Python's built-in web server to allow an attacker to gain arbitrary code execution on the host. It is always important to reduce the attack surface whenever possible, and accidentally exposing development features to a network is certainly not a best-practice. This vulnerability was fixed in August of 2010, and then unintentionally reintroduced in December of 2010 and went undetected until now.

It should be noted that the Python developers take a rather proactive stance on security; the codebase is scanned every other day using Coverity. The fact that I was able to identify bugs that were missed by Coverity indicates that there is some merit to my approach.

In addition to these vulnerabilities, some development metrics were calculated during the analysis process. First, Table 4.1 displays the top 10 developers with the highest vulnerability injection rate. This metric was calculated during the bug injection analysis. For each bug that we were able to trace back to its originating changeset, we are able to note the developer that injected the bug. The number of bugs injected is then compared to the total number of changesets committed by the developer, giving us the developer's vulnerability injection rate. This table ignores outliers that have less than three total commits.

Developer	Reports
Dev12	18
Dev44	10
Dev18	5
Dev45	3
Dev46	3
Dev47	3
Dev14	3
Dev24	3
Dev48	2
Dev30	2

Table 4.2: Python Developer Vulnerability Reports (Top 10)

Next we looked at the developers who have proven talented at finding and reporting security vulnerabilities. Table 4.2 lists the 10 developers who have submitted the most vulnerability reports. These developers should be encouraged to keep reviewing code and finding vulnerabilities. Perhaps they employ certain techniques from which other developers could benefit.

Finally, in Table 4.3 we list the developers who have shown skill at resolving vulnerabilities. This table may be a bit misleading, though, because occasionally vulnerabilities are patched by developers that do not have commit privileges to the main source code repository. In those cases, a core Python contributor must apply the patches, and they will be shown in the history as the developer that resolved the vulnerability. Of course this role is still important, because if the core developers were not vigilant about reviewing and applying patches that are ready, there could be large delays in resolving the vulnerabilities.

Because we were able to identify changesets that injected vulnerabilities, we were also able to calculate the amount of time that vulnerabilities existed in the code before they were reported and then resolved. Figure 4.1 graphs the time it took to fix the vulnerabilities in Python that we studied. Just as a point of comparison, Google Project Zero [44] is a security team funded by Google that is tasked with making the internet more secure. Project Zero contributes to the security of the internet by finding, reporting, and sometimes fixing high-impact vulnerabilities in popular software. They set a 90-day deadline for full-disclosure on

Developer	Resolutions
Dev14	37
Dev12	34
Dev31	24
Dev37	24
Dev36	21
Dev42	20
Dev18	14
Dev43	11
Dev24	11
Dev30	10

Table 4.3: Python Developer Vulnerability Resolutions (Top 10)

the bugs that they discover. In other words, once they report the vulnerability to the vendor, they give the vendor 90 days before publicly releasing the details of the vulnerability. By doing so, Project Zero asserts that it should not take longer than 90 days to fix a vulnerability once it has been identified.

Figure 4.2 displays the amount of time between when vulnerabilities were introduced into Python and when they were first reported to the bug tracker.

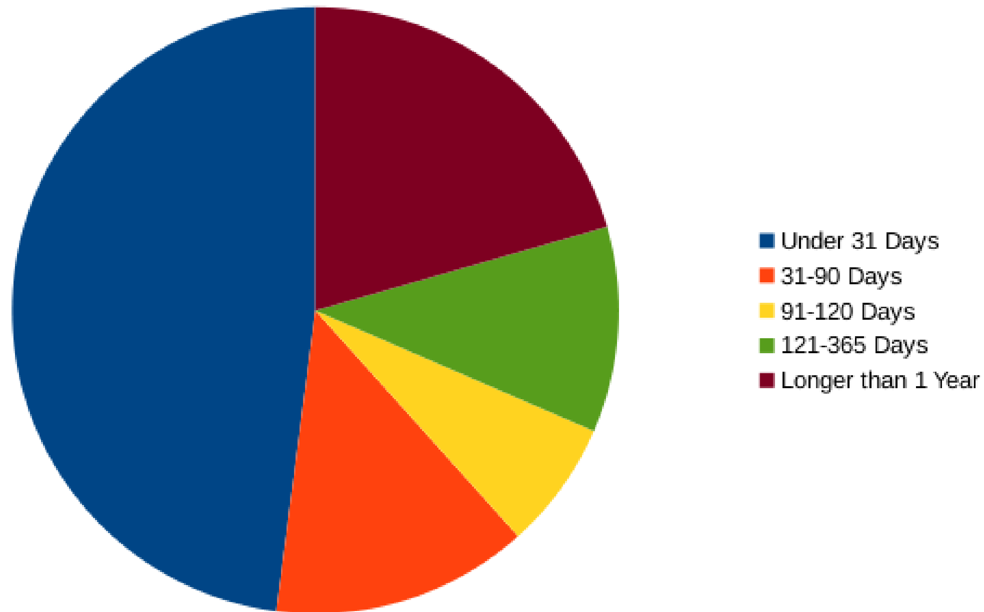
These graphs show that while vulnerabilities can often take a long time to discover, they are usually fixed relatively quickly.

While these statistics were collected using only publicly available data, the results were anonymized. It is important to note that these results are intended to be used by the development team to improve the overall security practices of the team, and should be used to coach developers and provide some constructive insight into ways they can look to improve their practices.

4.2 Roundup Results

The Roundup project is a bug tracker with a web interface. The Python project actually uses roundup as their bug tracking software. I performed an analysis on the roundup data set, but it was significantly smaller than the Python project. The repository containing the

Figure 4.1: Time To Fix Security Issues in Python

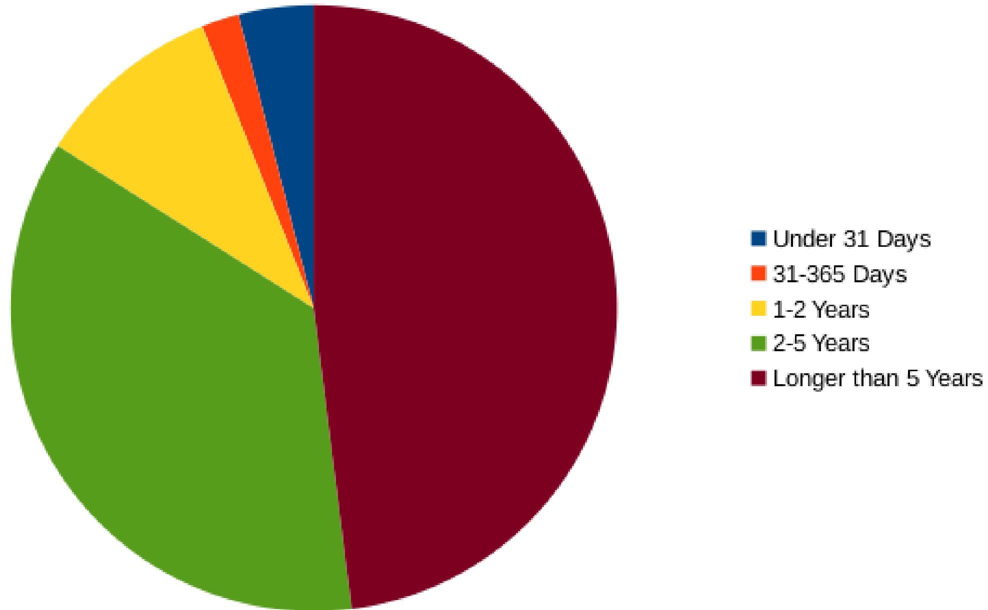


Roundup codebase shows only 4940 changesets, and their bug tracker only has 22 issues marked as security issues.

I wanted to demonstrate that it is possible to use these methods to detect bugs in smaller projects as well as large ones. The Roundup project is a pretty typical small open source project. According to Open Hub, the project has an average of less than 100 commits per month from an average of two developers. The project has been around since 2001 and has amassed around 100,000 lines of code. [19]

Because of its size, there was not enough training data to make any useful discoveries from the project itself. I did, however, use data gathered while testing the Python project. Four results were found, and upon examining them manually it appears that two of them are valid. They involve the use of unsafe Python functions, and I am in the process of reporting and patching the issues. The other two issues looked like directory traversal vulnerabilities, but there were mitigations in place to ensure files outside of allowed html directories were not served.

Figure 4.2: Time To Find Security Issues in Python



These results are exciting because even though there was not enough metadata available to find any meaningful results directly, we were still able to find results using the knowledge gained from another project. This accomplishment means that as more and more projects are analyzed using these techniques, our database of development patterns and vulnerable code signatures grows and can be used to find new vulnerabilities in any software project for which we have access to the source code, regardless of the amount of metadata available.

4.3 Vulnerability Injection Patterns

When extracting patterns from large amounts of data, sometimes patterns are found that cannot be explained or do not immediately make sense. Nevertheless, if we can then use those patterns to detect additional vulnerabilities, then it does not necessarily matter that they make sense to us. It is certainly outside the scope and separate from the goal of this experiment to divine meaning from every correlation identified.

The vulnerability injection patterns that were identified by this research confirm that bugs tend to have high spatial locality. [56] This information is useful because it can point

File	Issues
Objects/unicodeobject.c	16
Modules/_ssl.c	15
Lib/ssl.py	13
Lib/zipfile.py	11
Lib/platform.py	10
Lib/netrc.py	10
Lib/smtpd.py	9
Python/sysmodule.c	9
Lib/os.py	8
Python/mysnprintf.c	8

Table 4.4: Files in the Python Project Containing Security Issues (Top 10)

security analysts at areas of code that have frequent security shortcomings. For Python, Table 4.4 displays the top 10 files that have been found to contain past security issues. While these issues have since been fixed, there may still be other issues that have not yet been discovered. It is also possible that a fix can introduce a new security issue or reintroduce an old security issue. In the case of the latter, where old security issues are reintroduced, the methods outlined in this dissertation can be applied to discover and fix the bugs. Another common pattern these methods would be useful for identifying is the occasional duplication of code. If code has been copied and pasted to another location, and is later found to contain a security vulnerability, it would be helpful to be alerted to the fact that there is still vulnerable code that requires the same fix.

The files in Table 4.4 represent 22% of all security issues in the public bug tracker. In other words, one fifth of all security issues identified in the bug tracker involve bugs in one of these 10 files. Since there are over 2,000 source code files in the Python repository, it could be very useful to have a list of files that are known to be historically vulnerable.

4.4 False Positives

Results were validated manually by reviewing the code to determine which issues flagged were false positives and which issues were indeed actual security vulnerabilities (true positives). For the Python project, a total of 38 issues produced code signatures that could be used to try to identify new vulnerabilities. Of those 38 issues, 17 of them did not have any hits when the current version of the Python source code was searched. That left 21 issues with potential hits. Those issues were then manually verified in order to determine whether they were true or false positives. This manual verification only took approximately 15 minutes, and two issues were identified as true positives.

Therefore, our false positive rate was approximately 90%. Still, 90% of 21 issues is only 19, which is a very reasonable number of issues for a security analyst to manually review. From start to finish this process only required less than two hours, and approximately only 15 minutes of manual analysis by a security analyst was needed. The result was two previously undiscovered security vulnerabilities identified and fixed. It was easy to create a patch that resolved the vulnerabilities found because they were similar to previously resolved vulnerabilities, and the new vulnerabilities could be fixed in the same manner.

Direct comparison to existing static analysis tools is not an apples-to-apples comparison, but it can still give us some idea of how well this method performs in terms of the false positive rate. Existing static analysis tools only look at one version of the source code of the program being analyzed, while this method uses other development metadata in addition to the source code. This method is meant to be used in conjunction with existing static analysis tools, and has a different set of goals (namely, it aims to detect any bugs which may be similar in some way to ones previously seen and fixed). The developers of Coverity believe that a false positive rate greater than 30% causes problems for the users, where they may begin to ignore subtle true positives from the tool. [25] I argue that this number matters less when the total number of reported issues from the tool is low enough to work through in one sitting.

Project	LOC (Approx.)	Total Issues	Code Signatures	Candidates	True Positives
Python	976,413	162	38	21	2
Roundup	88,578	22	6	4	2

Table 4.5: Vulnerability Detection Rates

Table 4.5 summarizes the vulnerability detection rates in the Python and Roundup projects that were analyzed during this experiment. The total issues column shows the number of issues in the public bug trackers that are marked as security issues. For our testing, we are interested mainly in these types of bugs, but this restriction could potentially be relaxed in order to search for other types of bugs as well. The code signatures are lines or snippets of code that were automatically determined to be vulnerable by analyzing changesets that are determined to inject vulnerabilities. The candidates column shows the number of locations in the current version of the project's source code that were flagged as potentially vulnerable, and finally the true positives column shows the number of locations that turned out to actually be vulnerable after manual verification.

We can tell from Table 4.5 that there were some valid vulnerabilities identified by our techniques. The vulnerabilities detected in this manner can be used to supplement additional findings from more traditional static analysis approaches. Additionally, the vulnerabilities identified are subtle and less likely to have been caught by static analysis tools.

The ability to find vulnerabilities that are similar to bugs previously seen and fixed is useful because we already know that these patterns were determined to be undesirable in the code. Occasionally when new vulnerabilities are discovered and reported to a project's mailing list the first reaction is to discuss whether or not the bugs are in fact vulnerabilities, and whether or not they should be fixed. If this discussion has already taken place, developers can often just cite the previous bug reports and message threads and move directly on to resolving the vulnerabilities.

4.5 Summary

The results of this experiment show that it is indeed possible to identify previously unknown vulnerabilities automatically using information gathered from development metadata. The vulnerabilities identified in the Python project illustrate how it is possible to isolate vulnerability injection using tagged issues from the public bug tracker in combination with metadata from the project's software repository. After correlating the data in messages from the issue threads in the bug tracker with text found in the commit messages of the changesets in the repository, we were able to isolate specific commits as injecting vulnerable code. We were able to pull out the known-vulnerable code and use it to identify other similar sections of code in the current most version of the software.

Using the same vulnerable code found in the Python project, we were able to locate similar code in the Roundup project and flag it as potentially vulnerable. After manual review, we were able to determine that we had discovered two new vulnerabilities in Roundup that were similar to the vulnerabilities found in the Python project. This is a powerful finding. It means that we can leverage the knowledge gained from analyzing mature, popular software projects like Python to help identify and fix vulnerabilities in smaller, less mature software projects that may not have enough history or development metadata yet to do the analysis on from scratch. This finding is useful because it expands the pool of software we are able to scan without the need for those projects to have a large amount of metadata (or even version control history) available. We can find bugs using only the current version of the source code.

These methods for leveraging resolved vulnerabilities to identify new vulnerabilities are novel, and the ability to find any previously undiscovered vulnerabilities marks an improvement in the status quo. These methods supplement existing techniques instead of replacing them, so this technique could become another useful tool in developers' toolkits that can help developers write more secure code.

Chapter 5

Conclusions

The work detailed in this dissertation enables us to detect previously unknown security vulnerabilities. Resolving these vulnerabilities, and in some cases just knowing about them, results in more secure code. Additionally, it is easier to manage a vulnerability detection tool that can give developers live feedback when they try to commit new changes to the code.

There are also several "value adds" that we have shown. In addition to the ability to detect 0days, it is useful to provide managers and software development communities with valuable metrics about the development of their software. Here are a few examples of the types of development metrics that managers can generate using this method:

- developers with a habit of introducing vulnerabilities
- number of vulnerabilities found over time
- developers skilled at finding/resolving vulnerabilities
- average time to find and fix vulnerabilities

For example, Tables 4.1, 4.2, and 4.3 demonstrate some useful statistics from the development of the Python project. These tables have been anonymized.

In Table 4.1, we can see the developers ordered by their vulnerability injection rate. This rate is calculated by finding the number of changesets committed by a developer that contain a vulnerability and dividing that number by their total number of committed changesets.

Table 4.2 shows the number of issue reports that each user has filed that are tagged as security issues. This table shows which developers are likely good at catching vulnerabilities.

Table 4.3 shows the number of changesets committed by a developer that are marked as resolving a security issue. Because of the structure of Python's development community, this list may not be an accurate representation of developers that are particularly adept at resolving security issues. Sometimes patches are submitted to the bug tracker, but not applied to the actual Python codebase by the original author. Still, in other organizations this could be a useful metric for identifying developers that are skilled at resolving vulnerabilities.

Figure 4.1 displays the amount of time between when security issues were reported and when a commit was checked in marking the security issue as resolved. Nearly half of the reported security issues were resolved within a month of being reported.

When bugs are reintroduced or duplicated, there is also a lower barrier to overcome when proposing a fix, as the previous discussion can be cited as explanation for the new fixes. Sometimes this barrier can be a significant issue when trying to resolve a vulnerability. Developers may not believe that the vulnerability is actually exploitable [37], they may argue about implementation details for the fix, or they may even argue it should not actually be fixed for arbitrary reasons. [49] [39]

The work detailed in this dissertation demonstrates a novel approach for finding new vulnerabilities in source code. By analyzing the publicly available development metadata for the Python project, we have shown that it is possible to automatically identify the specific commits that injected vulnerable code, and then leverage that knowledge to identify similar vulnerabilities in not just the current version of that project's code, but also in code from other projects.

The specific contributions made by this dissertation include a more accurate method for determining fault injection and two methods of using old, resolved vulnerabilities to automatically find previously unknown (but similar) vulnerabilities. These methods involve the use of several techniques from different fields in computer science including the utilization of test suites, version control system bisection, machine learning, and code clone detection. These techniques can be applied almost entirely automatically, with only some minimal

verification required by a security analyst to determine which results are true positives. In some cases, it is possible to apply previous fixes to the new vulnerabilities, resulting in vulnerability resolution with minimal manual work.

Armed with these new approaches, software engineers and security analysts can leverage existing development metadata to find and resolve bugs in their software. The methods explained in this dissertation are novel and complementary to existing methods. This approach is intended to augment existing methods in an effort to improve the security of applications by discovering and fixing security vulnerabilities. Of course, these methods can also be used to detect other types of bugs, but we have shown that they do indeed work to identify previously unknown vulnerabilities.

Chapter 6

Future Work

As mentioned previously, the focus of this experiment is on finding security vulnerabilities. That is not to say this method could not be generalized to find other classes of software bugs. The method could be extended to also look into memory leaks, for example, with little additional work. By simply removing the restriction on the issues pulled from the bug tracking database, we could use the data from all bugs to find regressions and code reuse, and could train the classifiers to look for changesets similar to those that injected any of the bugs we were able to successfully isolate.

Clearly the false positive rate is an area that requires further improvement. While it is manageable at its current level, if we were to try to scale the search up to include all types of bugs (not just security issues), we would likely have far too many findings to manually verify.

Another possible use of this tool would be to allow it to interface with other tools. It could be one additional way that code can be checked for vulnerabilities before it is shipped. A tool like RASAr or ISA [57] could use it to identify potential vulnerabilities. Alternatively this tool could use other vulnerability detection tools as additional "signals" that can help make a more authoritative decision on the security of a piece of code. These methods could then be introduced into the development lifecycle of a project in order to help catch bugs as they are introduced in a way similar to how the WordPress developers have implemented suggestions from supernothing regarding commit hooks. Now, whenever a commit is made to certain WordPress repositories, the changeset is scanned for known-vulnerable code.

In the original proposal for this experiment, I spoke of performing AST matching in addition to simple text matching. The feedback from some of my committee members made

me reconsider this, but I still feel it may be fruitful to pursue this idea further at a later date. It did turn out not to be necessary for finding bugs, as several discoveries were possible using simple text matching and fuzzy text matching, but better matching could only help improve the ability to find similar bugs.

It should be possible to take this work one step further, and provide a method for automatically suggesting fixes for vulnerabilities identified in the codebase of a project based on similar vulnerabilities and their known fixes in the past. Previous work by Dr. Hafiz shows that it is indeed possible to apply security transformations on existing code in an automated fashion. [45] If we can identify areas of code that are frequently vulnerable, maybe we can proactively apply some preventative security transformations to ensure that any vulnerabilities discovered in the future have minimal impact, much like traditional defense in depth techniques. [45]

Perhaps there are other ways that this tool can help the field of computer science. I look forward to having the opportunity to explore those new ways, and plan on continuing this research.

Bibliography

[1] Algorithms inside search google. http://www.google.com/insidesearch/howsearchworks/algorithms.html.

[2] And the winner is... https://mail.python.org/pipermail/python-dev/2009-March/087931.html.

[3] Apache subversion. http://subversion.apache.org/.

[4] CERT ROSE checkers. http://rosecheckers.sourceforge.net/.

[5] CERT secure coding. http://www.cert.org/secure-coding/.

[6] Clang static analyzer. http://clang-analyzer.llvm.org/.

[7] Concurrent versions system - summary [savannah]. http://savannah.nongnu.org/projects/cvs/.

[8] Coverity development testing. http://www.coverity.com/.

[9] Git. http://git-scm.com/.

[10] Gnu binutils. http://www.gnu.org/software/binutils/.

[11] Hp fortify. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812.

[12] IDA pro. https://www.hex-rays.com/products/ida/index.shtml.

[13] Linux kernel bug tracker. https://bugzilla.kernel.org/.

[14] Mercurial SCM. http://mercurial.selenic.com/.

[15] OSVDB: open sourced vulnerability database. `http://osvdb.org/`.

[16] Python bugtracker. `http://bugs.python.org/`.

[17] The python programming language open source project on open hub. `https://www.openhub.net/p/python`.

[18] ROSE compiler infrastructure. `http://rosecompiler.org/`.

[19] The roundup issue tracker open source project on open hub. `https://www.openhub.net/p/roundup`.

[20] Splint home page. `http://www.splint.org/`.

[21] Tony Abou-Assaleh and Wei Ai. Survey of global regular expression print (grep) tools. Technical report, Citeseer, 2004.

[22] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[23] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[24] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *, International Conference on Software Maintenance, 1998. Proceedings*, pages 368–377, 1998.

[25] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[26] David Brumley. *Analysis and Defense of Vulnerabilities in Binary Code*. ProQuest, 2008.

[27] Sang Kil Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy (SP)*, pages 380–394, 2012.

[28] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265278, March 2011.

[29] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[30] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, page 238252, New York, NY, USA, 1977. ACM.

[31] Kevin Crowston, James Howison, and Hala Annabi. Information systems success in free and open source software development: theory and measures. *Software Process: Improvement and Practice*, 11(2):123148, 2006.

[32] Kevin Crowston, Kangning Wei, Qing Li, and J. Howison. Core and periphery in Free/Libre and open source software team communications. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences, 2006. HICSS '06*, volume 6, pages 118a–118a, 2006.

[33] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Information Security*, page 346360. Springer, 2011.

[34] Janez Demšar, Blaž Zupan, Gregor Leban, and Tomaz Curk. *Orange: From experimental machine learning to interactive data mining*. Springer, 2004.

[35] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.

[36] Saso Deroski and Bernard enko. Is combining classifiers with stacking better than selecting the best one? *Machine Learning*, 54(3):255–273, 2004.

[37] Chris Evans. The poisoned nul byte, 2014 edition. `http://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html`.

[38] Ming Fang and Munawar Hafiz. Discovering buffer overflow vulnerabilities in the wild: An empirical study. 2014.

[39] Alex Gaynor. Enabling certificate verification by default for stdlib http clients. `http://legacy.python.org/dev/peps/pep-0476/`.

[40] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013–Object-Oriented Programming*, pages 629–653. Springer, 2013.

[41] GNU. Grep. `http://www.gnu.org/software/grep/`.

[42] GNU. Diffutils. `http://www.gnu.org/software/diffutils/`, 2002.

[43] Mathieu Goeminne and Tom Mens. A framework for analysing and visualising open source software ecosystems. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, page 4247, New York, NY, USA, 2010. ACM.

[44] Google. Project zero. `http://googleprojectzero.blogspot.com/`, 2014.

[45] Munawar Hafiz. *Security on demand*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[46] Munawar Hafiz, Paul Adamczyk, and Ralph Johnson. Systematically eradicating data injection attacks using security-oriented program transformations. In *Engineering Secure Software and Systems*, pages 75–90. Springer, 2009.

[47] Christopher Harrison. Odinn: An in-vivo hypervisor-based intrusion detection system for the cloud. Unpublished manuscript, May 2014.

[48] Christopher Harrison, Devin Cook, John A Hamilton, and Robert McGraw. Dynamic binary analysis for vulnerability detection. Unpublished manuscript, May 2012.

[49] Andreas Hasenack. New ssl module doesn't seem to verify hostname against commonname in certificate. `http://bugs.python.org/issue1589`.

[50] James Howison, Megan Conklin, and Kevin Crowston. FLOSSmole: a collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[51] Nwokedi Idika and Aditya P. Mathur. A survey of malware detection techniques. *Purdue University*, page 48, 2007.

[52] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77131, 2007.

[53] Jorma Kajava, Juhani Anttila, Rauno Varonen, Reijo Savola, and Juha Rning. Senior executives commitment to information security from motivation to responsibility. In Yuping Wang, Yiu-ming Cheung, and Hailin Liu, editors, *Computational Intelligence and Security*, number 4456 in Lecture Notes in Computer Science, pages 833–838. Springer Berlin Heidelberg, January 2007.

[54] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[55] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[56] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[57] Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu. ISA: a source code static vulnerability detection system based on data fusion. In *Proceedings of the 2nd international conference on Scalable information systems*, InfoScale '07, page 55:155:7, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[58] U. Lindqvist and E. Jonsson. How to systematically classify computer security intrusions. In *, 1997 IEEE Symposium on Security and Privacy, 1997. Proceedings*, pages 154–163, 1997.

[59] Bing Liu Wynne Hsu Yiming Ma. Integrating classification and association rule mining. In *Proceedings of The Fourth International Conference on Knowledge Discovery and Data Mining*, 1998.

[60] David Maynor. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Elsevier, April 2011.

[61] Martin Michlmayr and Anthony Senyard. A statistical analysis of defects in debian and strategies for improving quality in free software projects. *The economics of open source software development*, page 131148, 2006.

[62] Idongesit Mkpong-Ruffin and David A Umphress. Software security. *Crosstalk*, 801:18–21, 2007.

[63] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, page 15, New York, NY, USA, 2005. ACM.

[64] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[65] J Overbey, Matthew D Michelotti, and R Johnson. Toward a language-agnostic, syntactic representation for preprocessed code. In *Proceedings of the 3rd Workshop on Refactoring Tools. ACM*, 2009.

[66] Jeffrey L Overbey. *A toolkit for constructing refactoring engines*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[67] Jeffrey L Overbey and Ralph E Johnson. Generating rewritable abstract syntax trees. In *Software Language Engineering*, pages 114–133. Springer, 2009.

[68] Jeffrey L Overbey and Ralph E Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In *ACM SIGPLAN Notices*, volume 44, pages 493–502. ACM, 2009.

[69] Daniel Quinlan and Thomas Panas. Source code and binary analysis of software defects. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, page 40:140:4, New York, NY, USA, 2009. ACM.

[70] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[71] John Ross Quinlan. *C4.5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[72] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[73] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: a new approach to computer security via binary analysis. In *Information systems security*, page 125. Springer, 2008.

[74] supernothing. Explo(it|r)ing the wordpress extension repos. `http://spareclockcycles.org/2011/09/18/exploitring-the-wordpress-extension-repos/`.

[75] Ha Manh Tran, Georgi Chulkov, and Jrgen Schnwlder. Crawling bug tracker for semantic bug search. In Filip De Turck, Wolfgang Kellerer, and George Kormentzas, editors, *Managing Large-Scale Service Deployment*, number 5273 in Lecture Notes in Computer Science, pages 55–68. Springer Berlin Heidelberg, January 2008.

[76] Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical computer science*, 92(1):191–211, 1992.

[77] D. Wahyudin and A.M. Tjoa. Event-based monitoring of open source software projects. In *The Second International Conference on Availability, Reliability and Security, 2007. ARES 2007*, pages 1108–1115, 2007.

[78] Sun Wu and Udi Manber. Agrep – a fast approximate pattern-matching tool. *Usenix Winter 1992*, pages 153–162, 1992.

Appendices

A. Code Listing

Appendix A

Code Listing

src/correlate.py

```
#!/usr/bin/env python

from __future__ import print_function

import re
import sys
import operator
import requests
from BeautifulSoup import BeautifulSoup
from dateutil import parser as dateparser
from time import sleep

import artifacts.code
from artifacts.bug import row_factory as bug_factory
from artifacts.bug import bug_type, resolution


db = None


def reverse_dict(d):
    return dict((v, k) for (k, v) in d.items())


def correlate():
```

```
global db, bug_type
if not db:
    raise Exception('must connect to db first')

rbug_type = reverse_dict(bug_type)
rresolution = reverse_dict(resolution)

# start with all bugs that are marked as security issues
bug_cursor = db.cursor()
bug_cursor.row_factory = bug_factory
bug_cursor.execute('select * from tracker_bugs where bug_type=? and ' +
                   '(resolution is NULL or resolution!=?)',
                   (rbug_type['security'], rresolution['not a bug']))

security_issues = {bug.id: bug for bug in bug_cursor}
bug_cursor.close()

# find all related messages
msg_cursor = db.cursor()
msg_cursor.execute('select * from tracker_messages order by date asc')
for msg in msg_cursor:
    # msg := (bug, author, message, date)
    try:
        security_issues[msg[0]].messages.append((msg[1], msg[2],
                                                 dateparser.parse(msg[3]))
                                                )
    except KeyError:
        #not interested in messages for that bug
        continue

commit_re = re.compile(r'(?<!\(default:)\b[0-9a-f]{12}\b', re.IGNORECASE)
for bug in security_issues.values():
    # find all referenced commits
```

```
        mentions = reduce(operator.or_,
                          map(set, (commit_re.findall(m[1])
                                    for m in bug.messages)))
        bug.references.update(('commit', m) for m in mentions)
        # set the author of the bug report
        # this is an int that points at a tracker user
        bug.reporter = bug.messages[0][0]

    issue_re = re.compile(r'issue ?(\d+)|#(\d+)', re.IGNORECASE)
    db.create_function('contains_issue', 1, lambda x: bool(issue_re.search(x))
        )
    commit_cursor = db.cursor()
    commit_cursor.execute('select revision, message from changesets '
                          'where contains_issue(message)')
    for commit in commit_cursor:
        # commit := (revision, message)
        for issue in issue_re.finditer(commit[1]):
            try:
                bug = security_issues[int(issue.group(1) or issue.group(2))]
                bug.resolved.add(commit[0])
                bug.back_links.add(('commit', commit[0]))
            except KeyError:
                # not a security issue
                pass

    commit_cursor.close()

    return security_issues


def resolve_commits(bugs, loc):
    artifacts.code.open_repo(loc)
    for bug in bugs.values():
```

```
        if bug.related and type(list(bug.related)[0]) != tuple:
            #looks like this bug has already been converted
            continue
        bug.references = set(artifacts.code.get_commit(c[1])
                             if c[0] == 'commit' else c
                             for c in bug.references).difference(set([None]))
        bug.back_links = set(artifacts.code.get_commit(c[1])
                             if c[0] == 'commit' else c
                             for c in bug.back_links).difference(set([None]))
        bug.introduced = set(artifacts.code.get_commit(c)
                             for c in bug.introduced).difference(set([None]))
        bug.resolved = set(artifacts.code.get_commit(c)
                           for c in bug.resolved).difference(set([None]))


def get_known_users():
    global db
    cursor = db.cursor()

    cursor.execute('select * from tracker_users')
    users = {user[0]: list(user) + [[]] for user in cursor}
    cursor.execute('select * from tracker_user_emails')
    for user, addr in cursor:
        users[user][-1].append(addr)

    return users



def resolve_reporters(bugs, loc=None, creds=None):
    '''resolve each bug.reporter id to a username

    loc: location to hit for each userid
         (parameterized string ready for .format())
```

```
creds: authentication credentials tuple('user', 'passwd')
'''
global db
if not db:
    raise Exception('must connect to db first')

cursor = db.cursor()
users = get_known_users()
newusers = []
newemails = []

for progress, bug in enumerate(bugs.values()):
    if bug.reporter is None:
        continue
    if type(bug.reporter) == int and bug.reporter not in users:
        sleep(.25)
        r = requests.get(loc.format(bug.reporter), auth=creds)
        page = BeautifulSoup(r.text)
        # user := (id, name, username, org, timezone,
        #           homepage, is_committer)

        try:
            tz = page.find('option', selected=True)['value']
        except TypeError:
            tz = None

        user = [
            bug.reporter,
            page.find(id='realname')['value'],
            page.find(id='username')['value'],
            page.find(id='organisation')['value'],
            tz,
            page.find(id='homepage')['value'],
```

```
            any('Is CommitterYes' in row.text
                for row in page.findAll('tr')),
        ]
        users[user[0]] = user[:]
        newusers.append(user)
        assert len(page.findAll('a',
                   href=lambda url: url.startswith('mailto'))) == 1
        email_el = page.find('a',
                             href=lambda url: url.startswith('mailto'))
        emails = [email_el.text]
        alternate_el = email_el.parent.parent.nextSibling.nextSibling.pre
        emails.extend(alternate_el.text.split())
        users[user[0]].append(emails)
        newemails.extend((user[0], email) for email in emails)

        cursor.execute('update tracker_bugs set reporter=? where id=?',
                       (bug.reporter, bug.id))

    bug.reporter = users[bug.reporter]

    complete = float(progress + 1) / len(bugs)
    c_width = int(70 * complete)
    print('\r [' + ('=' * c_width) + (' ' * (70 - c_width)) +
          '] {:.0%}'.format(complete), end='')
    sys.stdout.flush()

print()

if newusers:
    cursor.executemany('insert into tracker_users (id, name, username, '
                       '    org, timezone, homepage, is_committer) '
                       'values (?,?,?,?,?,?,?)', newusers)
if newemails:
```

```
        cursor.executemany('insert into tracker_user_emails (userid, email) '
                           'values (?,?)', newemails)

    db.commit()
    cursor.close()


if __name__ == '__main__':
    import sqlite3
    from pprint import pprint

    if len(sys.argv) != 2:
        print("Usage: {} database.db".format(sys.argv[0]))
        exit()

    db = sqlite3.connect(sys.argv[1])

    pprint(correlate())

    db.close()
```

src/analyze.py

```
#!/usr/bin/env python

from __future__ import print_function

#dependencies
import unidiff

#stdlib
import os
import re
import shutil
import tempfile
import subprocess
from StringIO import StringIO
from collections import Counter

#my code
import correlate
import artifacts.code
from train import train


db = None


MIN_GREP_LEN = 5
MAX_INJECTION_THRESHOLD = 5
CODE_EXTENSIONS = ('.py', '.c', '.h')
IGNORE = ('#', '//', 'import ', 'from ')
GREP_IGNORE = ('return True', 'return False', 'return NULL;',
               'return 0;', 'return 1;', 'break;')
```

```
TEST_RE = re.compile(r'^def (test_\w+)\(')


def find_cause(bug, method='unit tests'):
    '''returns a set of commits and a set of lines that may have caused the
        bug

    Currently supported methods include (in order of accuracy) "file history",
    "line history", and "unit tests" (very slow).

    Future: "assisted" mode?
    '''

    if not bug.related:
        return set(), set()

    #as a backup, use all related commits
    resolutions = list(bug.related)
    if bug.resolved:
        resolutions = bug.resolved

    candidates = set()
    bad_code = set()
    test_stats = {}

    for commit in resolutions:
        if method == 'file history':
            #for each file changed, find the last commit that changed that
                file
            for f in commit._changeset.changed():
                if 'test_' in f.path or not f.path.endswith(CODE_EXTENSIONS):
                    #ignore changes to unittest and non-code files
```

```
                continue

            try:
                index = f.history.index(commit._changeset)
                c = f.history[index + 1]
            except ValueError:
                #changeset missing from history list??
                #maybe file was renamed?
                pass
            except IndexError:
                #file has no older changes
                pass
            else:
                candidates.add(artifacts.code.Commit(c))

    elif method == 'line history':
        # this is a simplified method just to demonstrate the difference
        #  in approach
        diff_file = StringIO(commit.diff())
        patch = unidiff.parse_unidiff(diff_file)

        #for each line changed, find the last commit that changed that
            line
        for f in patch:
            if 'test_' in f.target_file or f.is_added_file \
               or not f.target_file.endswith(CODE_EXTENSIONS):
                #ignore changes to unittest and non-code files
                #skip added files
                continue

            bad_lines = []
            for hunk in f:
                #there are three types of changes:
```

```
        # line deletions - go back one version and find
            annotations
        # line changes - can treat same as deletions
        # line additions - look at the line before the addition
        #  for now just treat the last change to that area as bad
        if hunk.deleted or hunk.modified:
            lineno = hunk.source_start
            target_line_meat = map(str.strip, hunk.target_lines)
            for linetype, line in zip(hunk.source_types,
                                      hunk.source_lines):
                line = line.strip()
                if linetype == '-' and line \
                   and not line.startswith(IGNORE):
                    bad_lines.append(lineno)
                    if len(line) > MIN_GREP_LEN \
                       and not line.startswith(GREP_IGNORE) \
                       and line not in target_line_meat:
                        bad_code.add(line)
                lineno += 1
        else:
            #must have been only an addition
            lineno = hunk.source_start
            for linetype in hunk.target_types:
                if linetype == '+':
                    #look at the line before the addition
                    bad_lines.append(lineno - 1)
                    break
                lineno += 1

    #get the version from the previous revision
    prev = commit._changeset.parents[0]
    blame = list(prev.get_node(f.source_file[2:]).annotate)
    for lineno in bad_lines:
```

```
            #blame list is 0-indexed, line numbers are 1-indexed
            #blame line := (lineno, sha, commit_func, line_content)
            candidate = artifacts.code.Commit(blame[lineno - 1][2]())
            candidates.add(candidate)

elif method == 'unit tests':
    # this method is very slow
    if len(commit._changeset.parents) != 1 or \
       commit._changeset.branch != 'default':
        #ignore merges and stay on the default branch
        continue

    diff_file = StringIO(commit.diff())
    patch = unidiff.parse_unidiff(diff_file)

    # grab all the unittest files that were modified
    for f in commit._changeset.changed():
        if 'test_' not in f.path or not f.path.endswith(
            CODE_EXTENSIONS):
            #only interested in unittest files
            continue

        if commit.id not in test_stats:
            test_stats[commit.id] = {}

        #extract the changed unittest file
        tmp_id, tmp_name = tempfile.mkstemp(suffix='.py')

        # each test file entry is mapped to a list:
        #  [failing, [testcases]]
        test_stats[commit.id][tmp_name] = [
            commit._changeset.parents[0],
        ]
```

```
            os.write(tmp_id, f.content.encode('utf8'))
            os.close(tmp_id)

            #grab all the new test case names
            for patched_f in patch:
                if patched_f.path == f.path:
                    testcases = []
                    for hunk in patched_f:
                        if not hunk.added:
                            continue
                        for linetype, line in zip(hunk.source_types,
                                                  hunk.source_lines):
                            if linetype != '+':
                                continue
                            line = line.strip()
                            m = TEST_RE.search()
                            if m:
                                testcases.append(m.group(1))
                    test_stats[commit.id][tmp_name].append(testcases)
                    break

    else:
        raise ValueError('unknown method: {}'.format(method))

if method == 'unit tests':
    # this method is very slow
    # this script expects a build/test script at the root of your VCS repo
    # named "do_test"
    for bugfix_id, stats in test_stats.items():
        for testfile, stat_data in stats.items():
            for case in stat_data[1]:
                workdir = artifacts.code.repo.path
```

```
            testname = os.path.join(workdir, 'current_test.py')
            shutil.copy(testfile, testname)
            shutil.copy(os.path.join(workdir, 'do_test'),
                        os.path.join(workdir, 'run_test'))
            with open(os.path.join(workdir, 'run_test'), 'a') as
                test_runner:
                test_runner.write("\n./python current_test.py | egrep
                    '^{} .* ok$'\n".format(case))

            try:
                subprocess.check_call("cd {} && hg bisect -r".format(
                    artifacts.code.repo.path),
                                      shell=True)
                result = subprocess.check_output("cd {} && hg bisect -
                    c ./run_test".format(artifacts.code.repo.path),
                                                 shell=True)
            except subprocess.CalledProcessError:
                #bisect failed
                stat_data[0] = None
                break

            if result.startswith("The first bad revision is:"):
                result = result.splitlines()[1]
                cid = int(result.split(':').strip())
                stat_data[0] = artifacts.code.get_commit(cid)
            else:
                stat_data[0] = None

print('done! building candidate list')

for stats in test_stats.values():
    for stat_data in stats.values():
        if stat_data[0].id != None:
```

```
                candidates.add(artifacts.code.Commit(stat_data[0]))

        # clean up all our temp files
        for files in test_stats.values():
            map(os.remove, files)

    return candidates, bad_code



if __name__ == '__main__':

    import sys
    import sqlite3

    if len(sys.argv) != 3:
        print('Usage: {} <db> <vcs>'.format(sys.argv[0]))
        exit()

    db = correlate.db = sqlite3.connect(sys.argv[1])
    repo_dir = os.path.expanduser(sys.argv[2])
    artifacts.code.open_repo(repo_dir)
    bugs = correlate.correlate()
    correlate.resolve_commits(bugs, repo_dir)
    results = {}

    print('analyzing bugs...')

    for bug in bugs.values():
        if not bug.related:
            #skip the bugs we don't yet have references for
            continue

        #Identify the commit that introduced the bug:
```

```
    injected, bad_code = find_cause(bug)
    if injected and len(injected) <= MAX_INJECTION_THRESHOLD:
        print(bug, '->', len(injected), len(bad_code))
        bug.introduced.update(injected)
        results[bug] = bad_code

#find potential bugs by grepping for bad lines
output = open('results.txt', 'wb')
for bug, lines in results.items():
    if not lines:
        continue
    output.write('[*] {}\n'.format(bug))
    for line in lines:
        output.write('[ ] {}\n'.format(line))

    try:
        output.write(subprocess.check_output(['agrep', '-rF',
                                              '\n'.join(lines),
                                              repo_dir]))
    except subprocess.CalledProcessError:
        output.write('[!] no results found')

    output.write('\n\n')

output.close()

#calculate statistics
# developers with a habit of introducing vulnerabilities
# number of vulnerabilities found over time
# developers skilled at finding/resolving vulnerabilities
# average time to find and fix vulnerabilities
cursor = db.cursor()
users = correlate.get_known_users()
```

```python
user_lut = {}
for userid, user in users.items():
    for addr in user[-1]:
        user_lut[addr] = userid

stats_file = open('stats.txt', 'wb')
timeline_file = open('timeline.csv', 'wb')
injections = Counter()
reports = Counter()
resolutions = Counter()
files = Counter()
for bug in bugs.values():
    reports[bug.reporter] += 1
    for c in bug.introduced:
        try:
            injections[user_lut[c.committer]] += 1
        except KeyError:
            injections[c.committer] += 1
    for c in bug.resolved:
        try:
            resolutions[user_lut[c.committer]] += 1
        except KeyError:
            resolutions[c.committer] += 1
        for f in c.changed:
            files[f] += 1

    if bug.introduced:
        date_introduced = min(c.date for c in bug.introduced)
    else:
        date_introduced = ''

    date_reported = bug.messages[0][2]
```

```
    if bug.resolved:
        date_resolved = min(c.date for c in bug.resolved)
    else:
        date_resolved = ''

    timeline_file.write('{},{},{},{},\n'.format(bug.id, date_introduced,
                                                date_reported,
                                                date_resolved))

timeline_file.close()

stats_file.write('vulnerability injection rates\n')
db.create_function('get_userid', 1, lambda addr: user_lut.get(addr))
for user, num in injections.items():
    if isinstance(user, int):
        cursor.execute('select count(*) from changesets '
                       'where get_userid(committer)=?', (user,))
        total = cursor.fetchone()[0] or 1
        user = users[user][1]
    else:
        cursor.execute('select count(*) from changesets '
                       'where committer=?', (user,))
        total = cursor.fetchone()[0]
    s = u'{:.2%} {}\n'.format(float(num) / total, user)
    stats_file.write(s.encode('utf8'))

stats_file.write('\nvulnerability resolutions\n')
for user, num in sorted(resolutions.items(), key=lambda x: -x[1]):
    if isinstance(user, int):
        user = users[user][1]
    stats_file.write(u'{} {}\n'.format(num, user).encode('utf8'))

stats_file.write('\nvulnerability reports\n')
```

```
for user, num in sorted(reports.items(), key=lambda x: -x[1]):
    if isinstance(user, int):
        user = users[user][1]
    stats_file.write(u'{} {}\n'.format(num, user).encode('utf8'))

stats_file.write('\nfiles affected\n')
for f, num in sorted(files.items(), key=lambda x: -x[1]):
    stats_file.write(u'{} {}\n'.format(num, f).encode('utf8'))

stats_file.close()

#train classifiers
training_set = set()
for bug in results:
    training_set.update(bug.introduced)

classifier = train(training_set)
print('rules:\n', classifier.rules)

#find potential bugs using our classifier
candidates = []
for commit in artifacts.code.repo:
    if classifier.classify(commit):
        candidates.append(commit)

for changeset in candidates:
    print(changeset)
```

src/artifacts/code.py

```
from artifacts.base import BaseItem
import vcs

repo = None

def open_repo(repo_loc):
    global repo
    repo = vcs.get_repo(repo_loc)

def get_commit(c_id):
    global repo
    try:
        return Commit(repo[c_id])
    except vcs.exceptions.ChangesetDoesNotExistError:
        return None

class Commit(BaseItem):

    def __init__(self, changeset):
        super(Commit, self).__init__()
        #want to save it but expose a few fields
        self._changeset = changeset
        self.committer = changeset.committer_email
        self.date = changeset.date
        self.message = changeset.message
        self.diff = changeset.diff
        self.ids = set([changeset.id, changeset.raw_id, changeset.short_id])
        self.id = changeset.raw_id
        self.changed = changeset.affected_files
        self.parents = changeset.parents
```

```
    def __eq__(self, other):
        for id in self.ids:
            if id in other.ids:
                return True
        return False

    def __hash__(self):
        return int(max(self.ids, key=len), 16)

    def __repr__(self):
        return '<commit {}>'.format(self.id)


class Code(BaseItem):
    pass
```

src/artifacts/db.sql

```
pragma foreign_keys = ON;

create table email (
    fromaddr text,
    refs text,
    in_reply_to text,
    date datetime,
    message_id text primary key,
    subject text,
    body text
);

create table changesets (
    revision text primary key,
    committer text,
    date datetime,
    message text,
    diff blob
);

create table changed_files (
    revision text,
    filename text,
    foreign key(revision) references changesets(revision)
);

create table tracker_bugs (
    id integer primary key,
    status integer,
    severity integer,
```

```
    title text,
    assignee integer,
    superseder integer,
    bug_type integer,
    stage integer,
    resolution integer,
    priority integer,
    reporter integer
);

create table tracker_components (
    bug integer,
    component integer,
    foreign key(bug) references tracker_bugs(id)
);

create table tracker_keywords (
    bug integer,
    keyword integer,
    foreign key(bug) references tracker_bugs(id)
);

create table tracker_versions (
    bug integer,
    version integer,
    foreign key(bug) references tracker_bugs(id)
);

create table tracker_messages (
    bug integer,
    author integer,
    message text,
    date datetime,
```

```
    foreign key(bug) references tracker_bugs(id),
    foreign key(author) references tracker_users(id)
);

create table tracker_dependencies (
    bug integer,
    dependency integer,
    foreign key(bug) references tracker_bugs(id)
);

create table tracker_files (
    bug integer,
    fileid integer,
    foreign key(bug) references tracker_bugs(id)
);

create table tracker_nosy (
    bug integer,
    userid integer,
    foreign key(bug) references tracker_bugs(id)
);

create table tracker_users (
    id integer primary key,
    name text,
    username text,
    org text,
    timezone text,
    homepage text,
    is_committer boolean
);

create table tracker_user_emails (
```

```
    userid integer,
    email text,
    foreign key(userid) references tracker_users(id)
);
```

src/artifacts/bug.py

```
from artifacts.base import BaseItem

# negative indices mean we're unsure
status = { 1: 'open',
          -1: 'pending',
           2: 'closed',
          -2: 'languishing',
          None: None}

severity = {-1: 'critical',
             2: 'urgent',
             3: 'major',
             4: 'normal',
            -5: 'minor',
            None: None}

stage = { 3: 'needs patch',
          4: 'patch review',
          5: 'commit review',
          2: 'test needed',
          6: 'resolved',
         None: None}

resolution = { 2: 'duplicate',
               1: 'accepted',
               3: 'fixed',
              -3: 'later',
               6: 'out of date',
              -2: 'postponed',
               8: 'rejected',
```

```
            -1: 'remind',
            10: 'wont fix',
            11: 'works for me',
             4: 'not a bug',
            -5: 'third party',
            None: None}

priority = { 6: 'deferred blocker',
             4: 'normal',
             3: 'high',
             5: 'low',
             1: 'release blocker',
             2: 'critical',
            None: None}

bug_type = { 1: 'crash',
             2: 'compile error',
             3: 'resource usage',
             4: 'security',
             5: 'behavior',
             6: 'enhancement',
             7: 'performance',
             None: None}

def row_factory(cursor, row):
    b = Bug()
    (b.id, b.status, b.severity, b.title, b.assignee, b.superseder, b.bug_type
        ,
     b.stage, b.resolution, b.priority, b.reporter) = row
    b.resolve_fields()
    return b

class Bug(BaseItem):
```

```
id = None
status = None
severity = None
title = None
assignee = None
superseder = None
bug_type = None
stage = None
resolution = None
priority = None

# users
reporter = None
introducer = None
resolver = None

def __init__(self):
    super(Bug, self).__init__()
    self.messages = []
    self.introduced = set()
    self.resolved = set()

def __repr__(self):
    return 'issue{0.id}{{{0.status}|{0.bug_type}|msg:{1}|related:{2}}}'.
        format(self, len(self.messages), len(self.related))

def resolve_fields(self):
    self.status = status[self.status]
    self.severity = severity[self.severity]
    self.stage = stage[self.stage]
    self.resolution = resolution[self.resolution]
    self.priority = priority[self.priority]
```

```
        self.bug_type = bug_type[self.bug_type]


class RoundupBug(Bug):
    pass
```

src/artifacts/mail.py

```
from artifacts.base import BaseItem

from dateutil import parser as dateparser

def row_factory(cursor, row):
    e = Email()
    (e.fromaddr, e.refs, e.in_reply_to, e.date,
        e.message_id, e.subject, e.body) = row
    e.date = dateparser.parse(e.date)
    return e

class Email(BaseItem):

    fromaddr = None
    refs = None
    in_reply_to = None
    date = None
    message_id = None
    subject = None
    body = None
```

src/artifacts/base.py

```
# All data sources need to have some common metadata. This base class provides
#  the minimum required fields.

class BaseItem(object):

    def __init__(self):
        self.references = set()
        self.back_links = set()

    @property
    def related(self):
        return self.references | self.back_links
```

src/tools/import_bugs.py

```
#!/usr/bin/env python
# this tool reads roundup bugs via xmlrpc and drops it into the db
# Usage: ./import_bugs.py database.db roundup.example.com/endpoint filter
#        (filter argument must be a python dict)
#
# For the python bugtracker: bugs.python.org/xmlrpc

from __future__ import print_function

import sys
import sqlite3
import xmlrpclib

from time import sleep
from datetime import datetime

if len(sys.argv) != 4:
    print("Usage: {} database.db roundup.example.com/endpoint filter".format(
        sys.argv[0]))
    exit()

def tuplify(value):
    return (value,)

conn = sqlite3.connect(sys.argv[1])
db = conn.cursor()
db.execute('pragma foreign_keys = ON')

roundup_server = xmlrpclib.ServerProxy(sys.argv[2], allow_none=True)
bug_ids = roundup_server.filter('issue', None, eval(sys.argv[3]))
```

```
for progress, num in enumerate(bug_ids):
    bug = roundup_server.display('issue'+num)
    db.execute('insert into tracker_bugs (id, status, severity, title, ' +
               'assignee, superseder, bug_type, stage, resolution, priority) '
                   +
               'values (?,?,?,?,?,?,?,?,?,?)',
               (num, bug['status'], bug['severity'], bug['title'],
                bug['assignee'], bug['superseder'], bug['type'], bug['stage'],
                bug['resolution'], bug['priority']),
              )
    db.executemany('insert into tracker_components (bug, component) ' +
                   'values ({},?)'.format(num), map(tuplify, bug['components'
                      ]))
    db.executemany('insert into tracker_keywords (bug, keyword) ' +
                   'values ({},?)'.format(num), map(tuplify, bug['keywords']))
    db.executemany('insert into tracker_versions (bug, version) ' +
                   'values ({},?)'.format(num), map(tuplify, bug['versions']))
    db.executemany('insert into tracker_dependencies (bug, dependency) ' +
                   'values ({},?)'.format(num),
                   map(tuplify, bug['dependencies']))
    db.executemany('insert into tracker_files (bug, fileid) ' +
                   'values ({},?)'.format(num), map(tuplify, bug['files']))
    db.executemany('insert into tracker_nosy (bug, userid) ' +
                   'values ({},?)'.format(num), map(tuplify, bug['nosy']))

    for m_id in bug['messages']:
        msg = roundup_server.display('msg'+m_id)
        db.execute('insert into tracker_messages (bug, author, message, date)'
            +
                   ' values ({},?,?,?)'.format(num),
                   (msg['author'], msg['content'],
                    datetime.strptime(msg['date'],
                                      '<Date %Y-%m-%d.%H:%M:%S.%f>')),
```

```
                        )

    complete = float(progress+1)/len(bug_ids)
    print('\r [' + ('='*int(70*complete)) + (' '*(70-int(70*complete))) +
          '] {:.0%}'.format(complete), end='')
    sys.stdout.flush()
    sleep(1)

print()
conn.commit()
db.close()
conn.close()
```

src/tools/import_mail.py

```
#!/usr/bin/env python
# this tool reads mailman archive files and drops them into the db
# Usage: ./import_mail.py database.db archive1.gz ...

from __future__ import print_function

import sys
import gzip
import sqlite3

from mailbox import PortableUnixMailbox
from dateutil import parser as dateparser

def msgfields(msg):
    return (unicode(msg.get('from')),
            unicode(msg.get('references')),
            unicode(msg.get('in-reply-to')),
            dateparser.parse(msg.get('date')),
            unicode(msg.get('message-id')),
            unicode(msg.get('subject')),
            unicode(msg.fp.read().strip()),
           )

if len(sys.argv) < 3:
    print('Usage: {} database.db archive1.gz ...'.format(sys.argv[0]))
    exit()

conn = sqlite3.connect(sys.argv[1])
db = conn.cursor()

failed = []
```

```
files = sys.argv[2:]
for num, filename in enumerate(files):

    if filename.endswith('.gz'):
        f = gzip.GzipFile(filename, 'rb')
    else:
        f = open(filename, 'rb')

    mb = PortableUnixMailbox(f)

    messages = []
    for msgno, msg in enumerate(mb):
        try:
            fields = msgfields(msg)
        except:
            failed.append((filename, msgno))
            continue
        messages.append(fields)

    db.executemany('''insert or replace into email
                      (fromaddr, refs, in_reply_to, date,
                       message_id, subject, body)
                      values (?,?,?,?,?,?,?)''', messages)
    conn.commit()
    f.close()

    complete = float(num+1)/len(files)
    print('\r [' + ('='*int(70*complete)) + (' '*(70-int(70*complete))) +
          '] {:.0%}'.format(complete), end='')
    sys.stdout.flush()

db.close()
conn.close()
```

```
print()
if failed:
    print('failed to parse the following messages:')
    for id in failed:
        print(id)
```

src/tools/import_repo.py

```
#!/usr/bin/env python
# this tool reads changeset metadata from a repo and drops it into the db
# Usage: ./import_repo.py database.db /path/to/repo

from __future__ import print_function

import sys
import vcs
import sqlite3

if len(sys.argv) != 3:
    print("Usage: {} database.db /path/to/repo".format(sys.argv[0]))
    exit()

conn = sqlite3.connect(sys.argv[1])
db = conn.cursor()
db.execute('pragma foreign_keys = ON')

repo = vcs.get_repo(sys.argv[2])

for num, c in enumerate(repo):
    try:
        diff = unicode(c.diff())
    except UnicodeDecodeError:
        diff = c.diff().decode('latin1')

    changeset = (unicode(c.raw_id), unicode(c.committer_email), c.date,
                 unicode(c.message), diff)

    db.execute('insert or ignore into changesets ' +
               '(revision, committer, date, message, diff) ' +
```

```
                'values (?,?,?,?,?)', changeset)

        if db.rowcount:
            #new revision added, insert corresponding files
            db.executemany("insert into changed_files (revision, filename) " +
                           "values ('{}', ?)".format(c.raw_id),
                           [(f.decode('utf8'),) for f in c.affected_files])

        complete = float(num+1)/len(repo)
        print('\r [' + ('='*int(70*complete)) + (' '*(70-int(70*complete))) +
              '] {:.0%}'.format(complete), end='')
        sys.stdout.flush()

    print()
    conn.commit()
    db.close()
    conn.close()
```