Finding Bugs in Source Code Using Commonly Available Development Metadata

Devin Cook
Auburn University

Yung Ryn Choe
Sandia National Laboratories[*]

John A. Hamilton, Jr.
Mississippi State University

Abstract

Developers and security analysts have been using static analysis for a long time to analyze programs for defects and vulnerabilities. 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 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 additional pieces of information available that would prove useful for finding bugs in code, 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 paper 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 new vulnerabilities in large and small software projects by finding new vulnerabilities in the cpython and Roundup open source projects.

1 Introduction

This paper showcases some of the benefits of utilizing additional metadata 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.

Through two case studies using test data from cpython and Roundup, we demonstrate that it is possible 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.

[*]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.

2 Related Work

There are already some existing methods of finding vulnerabilities in software. There is also past research on the use 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 our methods.

Static analysis has been practical since the 1970s with pioneers such as Cousot [16] leading the way. Today there are a number of existing static analysis tools in use including some that are free software such as rosecheckers [2], Clang Static Analyzer [3], splint [10], and FindBugs [13]. In addition to many of these more "academic" and open source tools, there are also several proprietary tools available like Coverity [4], MAYHEM [15], and Fortify [5]. These tools are already used in the real world on a daily basis to find vulnerabilities.

2.1 Leveraging Development Metadata

There has been other research related to mining development metadata. However, it generally focuses too narrowly on only input, and makes little attempt to associate multiple sources of information. That means it examines *only* the data in a software repository, or *only* the data in a bugtracker. Livshits and Zimmermann created a tool called DynaMine that mines the revision history of a software repository in order to find common patterns and bugfixes that may help identify bugs [23]. Again, while they were quite successful, they did not utilize development data from other sources.

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 [28]. 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 [21].

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 [21].

2.2 Text Matching

Exact text matching is very straightforward. We can use a simple tool such as grep [17] 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 [29].

There are a number of tools for performing approximate text matching [11] including agrep [30]. 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 [30]. Since agrep is mostly compatible with grep's syntax, it is usually possible to use agrep as a drop-in replacement for grep. We use agrep to perform approximate text matching for our code clone detection method.

2.3 Token Matching

Research from Kamiya et al. details a manner of performing code clone detection. Their method, named CCFinder [20], 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 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.

2.4 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 [14].

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 [26].

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.

3 Experimental Design

In order to find new vulnerabilities, 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 project. We will accomplish this by using a combination of data aggregation, machine learning, and code clone detection techniques.

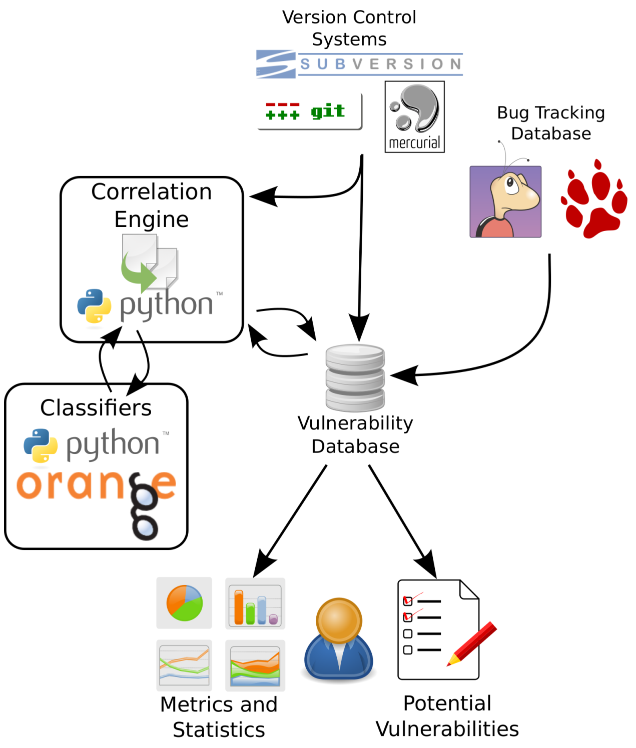

Figure 1: Project Architecture

With the entire project history available, it is often possible to go directly to the source code and find the actual commit that resolved the vulnerability being analyzed and therefore identify 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.

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 we 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, we 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 we will show that they can be used to identify previously unknown vulnerabilities in the current code base for an application.

3.1 Bug Injection Identification

After collecting all the necessary data, we identify a set of "related changesets" and "resolving changesets" for each bug. Identifying the changesets that have injected a particular bug is accomplished via two methods. The first method is an improvement on the SZZ algorithm [28] as outlined by Kim et al. [21]. 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.

3.1.1 Find Injection Using VCS Annotation

Version control systems provide a great benefit to development by allowing users to track the history of a project. 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”.

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 [21]. They call this tree an annotation graph, and find that using it improves their accuracy. Additionally, they exclude changes to blank lines and comments, make an effort to exclude format changes that make no semantic different, and exclude very large “bugfix” changesets.

3.1.2 Find injection using added test cases and VCS bisection

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 [25].

The Python project requires 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. Even for projects without this requirement, there still may be some unit tests submitted occasionally when corresponding vulnerabilities are fixed. It is possible to automatically identify those unit tests by correllating them with the bug resolutions and use them to help identify where the vulnerability was actually injected. 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. We can then 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 after this revision. This search is a binary search, and is `O(log n)` in time complexity with respect to the number of revisions in the source code repository. Note that this entire process is automated.

3.2 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 paper 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. The next step is to actually perform that identification.

3.2.1 Text Matching

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 [17] for those snippets. The text matching method is most useful for catching two types of bug injection: vulnerable code reuse and vulnerability reintroduction (regression). 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.

For our purposes, this code clone detection technique was sufficient to demonstrate the viability of our prototype. However, moving forward we would like to take advantage of some of the more advanced techniques like token and AST matching to improve our ability to detect other more complicated vulnerabilities.

3.2.2 Machine Learning

We can also utilize the data to discover new vulnerabilities by applying machine learning techniques to find patterns and identify some indicators of the likelihood that a changeset introduces a vulnerability. The main machine learning algorithms that are used in this experiment are C4.5 [27] and Association Rule Mining [12]. 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 [24].

A number of features including the committer’s bug injection rate, appearance of known-vulnerable code, and code change locations are used to train classifiers on the set of commits known to have introduced a vulnerability. These classifiers are used to detect commits similar to those known to have injected vulnerabilities.

3.3 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. This step is the only manual step. Up to this point, all previous tasks can be completed in an automated fashion.

Typically manual verification 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.

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. Keep in mind that even though a bug may not be exploitable, it may still be considered a valid bug and should likely still be fixed. Perhaps the current mitigations in place could be removed or altered in the future, which may create a condition where the previously mitigated vulnerability may become exploitable.

3.4 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 described in this paper to demonstrate these detection methods do not automatically provide resolutions in the form of a patch, it should sometimes be possible to do so [19]. 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 sometimes be possible to automatically provide a patch that reproduces the modifications made to fix the similar vulnerability previously.

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:

- Can we find new vulnerabilities?
- 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. Furthermore, once the results have been evaluated manually, they should not need further evaluation on subsequent runs. An analyst should only need to look at *new* findings.

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 paper 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 we used for evaluation was the Python project, or more precisely, cpython. 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 [6]) in 2009 [1] and have used a version of Roundup as the public bug tracking system that currently contains over 34,000 issues [7]. 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. The codebase is nearly 1 million lines of code as counted by Open Hub [8].

Our testing discovered two new valid vulnerabilities that we were able to easily report and resolve:

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

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. The server was configured to listen on 0.0.0.0 instead of only listening on the loopback address, thereby making it accessible to anyone on the same network. This vulnerability was fixed in August of 2010, and then unintentionally reintroduced in December of 2010 and went undetected until now.

In addition to these vulnerabilities, some development metrics were calculated during the analysis process. The vulnerability injection rates of all the developers were calculated, and this distribution was found to be non-uniform. Some developers appear more likely than others to introduce security vulnerabilities. We also looked at the developers who have proven talented at finding, reporting, and resolving security 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. About half of the vulnerabilities found required less than a month to fix after being reported. Nearly a quarter took longer than one year after being reported. Just as a point of comparison, Google Project Zero [18] is a security team funded by Google that is tasked with making the internet more secure. They set a 90-day deadline for full-disclosure on 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. Two thirds of the cpython vulnerabilities were fixed within 90 days.

We can also calculate the amount of time between when vulnerabilities were introduced into cpython and when they were first reported to the bug tracker. Interestingly, it took longer than 5 years to identify nearly half of the vulnerabilities studied.

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

4.2 Roundup Results

The Roundup project is a bug tracker with a web interface. We performed an analysis on the roundup data set, but it was significantly smaller than the Python project. The repository containing the Roundup codebase shows only 4940 changesets, and their bug tracker only has 22 issues marked as security issues. 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 [9].

Because of its size, there was not enough training data to make any useful discoveries from the project on its own. We did, however, use data gathered while testing the Python project. Because some of our code signatures from the cpython project included Python code, and because Roundup is written in Python, we were able to use code signatures identified from cpython. Using more sophisticated code clone detection techniques like AST matching theoretically means that we could use a project written in a different language for the training step.

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 we are in the process of reporting and patching the issues. 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

The vulnerability injection patterns that were identified by this research confirm that bugs tend to have high spatial locality [22]. This information is useful because it can point security analysts at areas of code that have frequent security shortcomings. For Python, Table 1 displays the top 10 files that have been found to contain past security issues. While these issues have since been fixed,

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

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 paper 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.

The files in Table 1 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. Obviously files inherently related to security (like the ssl module) will be represented here, but there are also other files with a high number of previous security vulnerabilities. Either way, since bugs have been shown to have high spatial locality [22], these files warrant extra scrutiny when looking for security vulnerabilities.

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 was needed. 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.

Table 2 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 2 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. We were able to isolate specific commits as injecting vulnerable code, and were then able to pull out the known-vulnerable code and use

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 2: Vulnerability Detection Rates

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. These methods supplement existing techniques instead of replacing them, so this technique could become another useful tool in developers' toolkits to help produce more secure code.

References

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

[2] Cert rose checkers. http://rosecheckers.sourceforge.net/.

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

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

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

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

[7] Python bugtracker. http://bugs.python.org/.

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

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

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

[11] ABOU-ASSALEH, T., AND AI, W. Survey of global regular expression print (grep) tools. Tech. rep., Citeseer, 2004.

[12] AGRAWAL, R., SRIKANT, R., ET AL. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (1994), vol. 1215, pp. 487–499.

[13] AYEWAH, N., HOVEMEYER, D., MORGENTHALER, J., PENIX, J., AND PUGH, W. Using static analysis to find bugs. *IEEE Software 25*, 5 (2008), 22–29.

[14] BAXTER, I., YAHIN, A., MOURA, L., SANT'ANNA, M., AND BIER, L. Clone detection using abstract syntax trees. In *, International Conference on Software Maintenance, 1998. Proceedings* (1998), pp. 368–377.

[15] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy (SP)* (2012), pp. 380–394.

[16] COUSOT, P., AND COUSOT, R. 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* (New York, NY, USA, 1977), POPL '77, ACM, p. 238252.

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

[18] GOOGLE. Project zero. http://googleprojectzero.blogspot.com/, 2014.

[19] HAFIZ, M. *Security on demand*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[20] KAMIYA, T., KUSUMOTO, S., AND INOUE, K. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on 28*, 7 (2002), 654–670.

[21] KIM, S., ZIMMERMANN, T., PAN, K., AND WHITEHEAD, E. J. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on* (2006), IEEE, pp. 81–90.

[22] KIM, S., ZIMMERMANN, T., WHITEHEAD JR, E. J., AND ZELLER, A. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 489–498.

[23] LIVSHITS, B., AND ZIMMERMANN, T. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2005), ESEC/FSE-13, ACM, pp. 296–305.

[24] MA, B. L. W. H. Y. Integrating classification and association rule mining. In *Proceedings of The Fourth International Conference on Knowledge Discovery and Data Mining* (1998).

[25] MKPONG-RUFFIN, I., AND UMPHRESS, D. A. Software security. *Crosstalk 801* (2007), 18–21.

[26] NEAMTIU, I., FOSTER, J. S., AND HICKS, M. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories* (New York, NY, USA, 2005), MSR '05, ACM, p. 15.

[27] QUINLAN, J. R. *C4.5: programs for machine learning*, vol. 1. Morgan kaufmann, 1993.

[28] ŚLIWERSKI, J., ZIMMERMANN, T., AND ZELLER, A. When do changes induce fixes? *ACM sigsoft software engineering notes 30*, 4 (2005), 1–5.

[29] UKKONEN, E. Approximate string-matching with q-grams and maximal matches. *Theoretical computer science 92*, 1 (1992), 191–211.

[30] WU, S., AND MANBER, U. Agrep – a fast approximate pattern-matching tool. *Usenix Winter 1992* (1992), 153–162.