Predicting Security Vulnerabilities in Software Components through Code and Meta-level Analysis

Roca He
Zhejiang University
heboyuan@gmail.com

Josiah Matlack
Northwestern University
jmatlack
@u.northwestern.edu

Chao Shi
Northwestern University
chaoshi2012
@u.northwestern.edu

Maciek Swiech
Northwestern University
maciejswiech2007
@u.northwestern.edu

ABSTRACT

In this paper, we present our research and findings toward an automated vulnerability analysis tool, which uses existing databases and code change logs, as well as PDG construction and dependency analysis, to predict undiscovered vulnerabilities in target software. Through semantic-level understanding of code, our aim is to apply a deeper analysis than provided by tools currently on the market.

1. INTRODUCTION

Software vulnerabilities are security-related bugs software that may be exploited by threats. Vulnerabilities are known to cost millions or billions of dollars to the affected vendors and users (Telang and Wattal 2005). Numerous methods have been used in practice to detect and eliminate vulnerabilities from software. These include manual code auditing, automatic testing, and static analysis. Despite years of research on eliminating vulnerabilities however, the number of vulnerabilities is still on the rise (Anon.).

In this proposal, we take a different approach towards vulnerability mitigation. With several years of software development and maintenance, we believe there is enough data to guide prediction of vulnerabilities in a meaningful manner. Intuitively and at a high-level perspective, studying code that was found to be vulnerable in the past may help to predict currently unknown vulnerabilities. That is the code in a previously vulnerable site and that in another site with a similar but yet unknown vulnerability is similar in some sense. Code similarity for such purposes may be measured in the forms of the dependencies of the components or pieces of code, or the system calls used, or other specific features such as checks of specific error numbers, and so on.

For this project, we will apply modern machine learning and data mining techniques on several security-focused features such as security advisories, alerts, and release notes, and also features from static analysis that we will develop. Specifically, for static analysis, we believe that control-flow and data-flow based analysis that reveals semantic-level artifacts of the code may be especially helpful. Furthermore, we will also use the version control history to obtain patterns relating different security advisories and alerts, results from static analysis, and possibly even coding styles. Previous work (Neuhaus et al. 2007; Neuhaus and Zimmermann 2009) has shown success at using component dependencies for predicting vulnerabilities. We will therefore also use dependencies amongst components as features in our system. While the security advisories and alerts and version control data are the meta-level information that hint at the pieces of code at which to focus, the code-level information obtained from dependency analysis and static analysis will provide us with a further understanding of code fragments that may be specially vulnerable.

Apart from the use of various features as described above, we will also use multiple machine learning and data mining techniques (including decision trees, neural networks, support vector machines, and association mining) and finally develop ensemble models based on 1these individual techniques to derive the best results. We expect to obtain from our model actionable insights that will speed up vulnerability discovery by directing related effort on specific areas in the code.

2. WORK

As we have briefly discussed in earlier sections, our system is divided into three main parts. These parts are the collection of vulnerability patch metadata, dependency analysis of affected code, and the analysis of program dependence graphs for affected code. This section describes each of these techniques in greater detail.

2.1 Metadata Collection

In order to collect data on common vulnerabilities and their corresponding fixes, we surveyed the Mozilla Foundation Security Advisories (MFSA) database. This is a database that includes all known vulnerabilities that affect Mozilla prod-

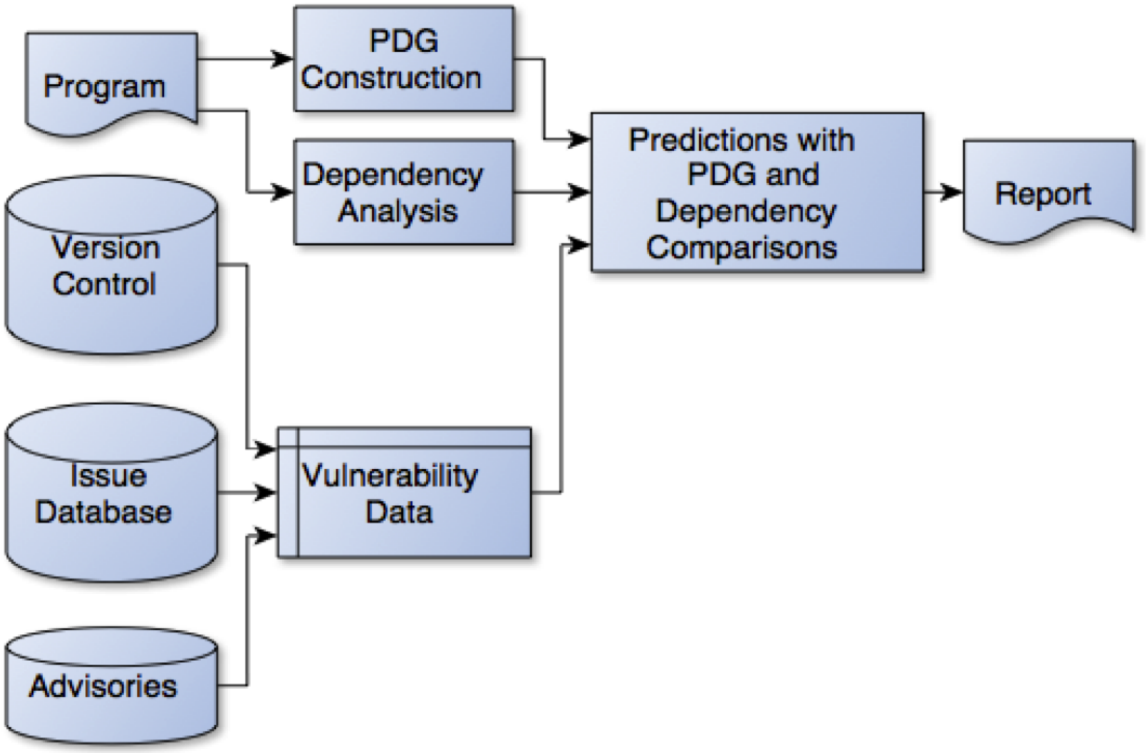

Figure 1: A diagram of the proposed system.

ucts and instructions on how users can protect themselves. Each commit to the code base has a BugID, and Each report has a unique MFSA ID associated with one or more Bug IDs.

We downloaded the code base using Mercurial, and dumped all the corresponding change logs. Then, each MFSA report was mapped to change sets or source code files using the Bug ID in the description of the change log. We used this database of information to find vulnerabilities.

2.2 Dependency Analysis

First, we need to statically extract all the imported libraries and function calls from the source files. To do this, less effort is needed than completely parse the source code and build the abstract syntax tree as expected in the program dependency analysis.

To extract the libraries imported by the source file, the parser will read the source code line by line and match the pattern of #include "xxx". The headers from the standard libraries in the format of #include "xxx" are not what we are interested. Besides, the C and C++ has compilation dependencies such as "# ifdef" which makes the dependency analysis much more complex. For the time being, we are not considering any of these macros.

For the function call extraction, we are matching the patter of "id (id, id, ...)". Taking the idea from Neauhaus paper, the reserved identifiers such as "if" or "for" are excluded. Besides, we are also trying to extract the functions defined and implemented by a single source file. Therefore, other source files calling a specific function can locate which source file the function is defined and implemented. Since the code base is pretty large and there are many identical naming of files, this method is based on the best effort and is by no means completely accurate.

We are examining the code base of Mozilla, which contains altogether 13740 C or C++ source files. The process is pretty fast and is terminated within 4 minutes.

After mining the Mozilla CVS and security advisories, we find altogether 256 vulnerabilities and 556 vulnerable components out of 13740 components. The next target is to examine what combination of the imported libraries and function calls will have high potential to cause vulnerability in the source code.

Different from the practice in Neauhaus paper, we are considering the set of the libraries as a single feature. If a file imported N libraries denoted as set S, then it will have $2^N - 1$ features, which comprises the super set of S except the empty set. For example, if a source file imported headers A.h, B.h, C.h. Then the source file will have 7 features, namely A.h,B.h, C.h, A.h, B.h, A.h, C.h, B.h, C.h and A.h, B.h, C.h. However, we notice that the number of the features will grow up exponentially as the size of library headers grows. To deal with this problem, we require that the number of the library headers a feature contains to be 3.

Next, we will have to filter the features; the feature size is still large even we limit the number of libraries in the fea-

Table 1: Survey of vulnerable components

Feature	Vulnerable	Non-vulnerable
nslviewManager.h nslDocument.h nsFocusManager.h	16	1
nslPresShell.h nsFrameManager.h nsLayoutUtils.h	15	3
jsXXX.h Marking.h	20	1

ture. Some combination of the headers will never appear in the Mozilla code base or they simply appear in the non-vulnerable components. Therefore, we require that the feature should appear at least in 3vulnerable source files. After this filtering, we will have 729 such features.

These features are appearing quite rarely in the non-vulnerable components but they do appear. We found 3956 non-vulnerable components out of 13740 which see at least one of these 729 features. The total appearing of features among these 3956 non-vulnerable components is 21700 with each non-vulnerable component seeing around 5.5 features individually. On the contrary, the 556 vulnerable source files are altogether seeing 15826 appearing counts, with each one seeing 28.5 on average. Some interesting findings about the features are summarized in table below.

We see that some of these libraries combined can produce some insights. For example, the first combination might be related to graphic components with the focusing capability. The second combination might be vulnerabilities coming from the frame components which have layout features. The third combination is seen in a lot of places, with one of the headers is Marking.h and the other two are all JavaScript headers. Although the size of instances is not large enough, the majority of this combined features are from the vulnerable components.

For mining on the libraries and the function calls, the python scripts are written and the information is stored in the database server run by Mysql. Some of the data process part is done manually by typing sql commands to fabricate the training samples and validation samples. For the machine learning algorithm, the library of LIBSVM is used.

2.3 Program Dependence Graph Analysis

In predicting software vulnerabilities, we require some basis of comparison between vulnerable code that has subsequently been patched, and the code being tested for analysis. The main technique we employed for this comparison was the construction and comparison of program dependence graphs (PDG). A PDG is a directed graph with vertices representing the statements of a program and the edges representing data, control, and alias flows.

Initially, we attempted to construct PDGs using libclang, a library of Python bindings for the LLVM compiler. LLVM provides various utilities for program analysis of C++ code, including some rudimentary mechanisms for creating control flow graphs and abstract syntax trees. After one of our weekly progress meetings, our team and Professor Chen came to the conclusion that, even with the utilities provided by LLVM, analysis of C++ code would be prohibitively hard, especially considering the short time frame of this project. This stems from the variability in C++ libraries and huge number of language features that it provides, and was made abundantly clear after analysis of various networking vulnerabilities that had been discovered in the Android operating system.

Table 2: Database Mining Results

	C	C and C++
Number of change sets	404,650	404,650
Number of C/C++ related bugs	84,138	152,394
Number of MFSA reports	542	542
Number of C/C++ vulnerabilities	132	256

After this point, we shifted our focus to analysis of C code. Toward this end, we discovered a tool for automated C language PDG construction, called Frama-C. Frama-C allows for the creation of PDGs, and outputs the information in both a text format and a visual representation created using graphviz. Using this tool, we were able to create PDGs for various C programs. In addition, we developed a Python script, adapted from our original work with libclang, that allowed for the parsing of the Frama-C output into a series of interconnected node objects. This script facilitated easier comparison of PDGs than was available with the original output from Frama-C (our original method of comparison was basically performed by hand).

3. EVALUATION

3.1 Finding Vulnerability Examples

We discovered many vulnerabilities through the use of the MFSA database. One example of such a vulnerability is the SSL certification validation vulnerability. Two options used by cURL in SSL verification are vulnerable to authentication errors if their values are set incorrectly. These values are

CURLOPT_SSL_VERIFYPEER
CURLOPT_SSL_VERIFYHOST

Setting these options to '0' ('false') or '1' ('true') wil skip certificate validation. The only valid option is '2'.

Another vulnerability found is also related to SSL certificates. In this, the gnutls_certificate_verify_peers2 function in the GnuTLS library may return 0, indicated an unsuccessful validation. However, this return code is not checked for this value, so errors may go by without identification.

In addition to these errors, we also found a few simple buffer overrun errors, and a few miscellaneous XSS bugs. The results of the database scan are shown below.

3.2 Dependency Analysis

After getting the vulnerable components and the non-vulnerable components sharing a set of features, we will be able to use machine learning techniques to classify new instances. The machine learning techniques we are using is support vector

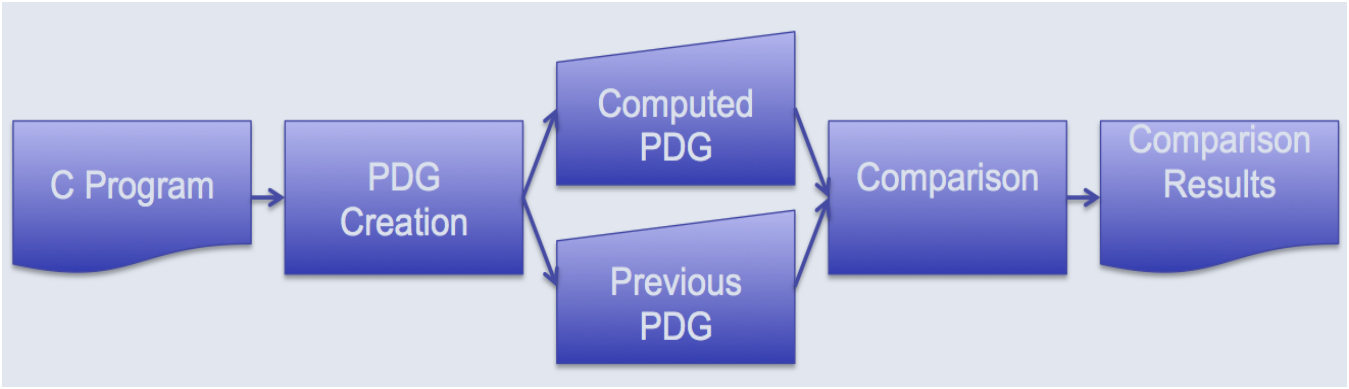

Figure 2: A diagram of the PDG construction process.

machine. The soft margin support vector machine variation is used with the variable of C set to be 1 as standard.

Unfortunately, for this stage, we will not be able to predict new vulnerabilities. The reason is that (1) Different from other machine learning task, which is easy to generate new data, this approach is highly dependent on the code base. Unless there are new components added to the code base, there are no good samples to test our machine learning algorithm. (2) The label of the new components is hard to determine. There is no fast way of generating the vulnerability label for the new components (Hard to do bug or vulnerability proof) while the support vector machine classifier can be very fast and work in large data set. This unbalance between the data generation and learning algorithm further limits our application.

Based on these limitations, we will use k-fold cross-validation to show at least that the dependency relationship can be learned to some extent. K is set to be 3 in this case, with the ratio between the training set and validation set to be 2:1.

After our random evaluation for 1000 times, the average false positive rate is 51.7field. Of all the components we believe that is vulnerable, only half of them are vulnerable in the real case. One belief which can relieve us is that some of the components are vulnerable but are considered non-vulnerable for the time being because people have not discovered them. That means the false positive rate is an upper bound and the real value should be lower than this.

On the other hand, the false negative rate is only 0.37small. Nearly all the vulnerable components can be successfully captured using the dependency analysis approach.

3.3 Lynx Libraries

Unfortunately, we were unable to meaningfully test the Lynx libraries using our PDG analysis, owing to multiple problems between the Frama-C framework and the Lynx library code. Frama-C appears to have issues with header files not included in the same directory as the file being analyzed. Frama-C uses gcc to pre-process the code, and so requires all symbols and libraries and calls to be explicated. Frama-C seems to be unable to deal with the complexity of elements such as GCC builtins, and C library code. We tried for many hours to debug these problems, but were unable to resolve errors such as:

```
/usr/include/bits/byteswap.h:47:[kernel]
warning: Calling undeclared function
\_\_builtin\_bswap32. Old style K&R code?
```

In this case, Frama-C is claiming that there are errors in standard C library code. Despite our best research efforts, these issues remain unresolved. Hopefully, with future input, these issues will be cleared up.

3.4 Test Programs

Our PDG construction was run on two small test programs, which can be found in the repository under data-ctrl.c and pdg-ex.c. The corresponding PDG outputs can be found in small-out and big-out. These graphs are not presented here due to the sheer size. Comparison analysis was not run on these small programs, because we have no existing corpus of possible vulnerabilities with which to compare them. However, the comparison tool can still be run on these files, if there is interest.

These small files were encouraging, as they provided proof-of-concept for the PDG generation, and also allowed for some calibration and testing of further scripts and routines that we attempted. Forming a playground of sorts, these programs allowed for rapid test iteration and bug fixes in our framework code. However, as test programs, and with the lack of meaningful comparison, these programs did not provide us with much research information.

4. RELATED RESEARCH

Attempts to quantify software reliability have been made since the 70s (Musa, Iannino, and Okumoto 1987). Many models have been developed that try to relate software faults to the age of the software, code complexity, programmer experience, organizational maturity, and so on (Takahashi and Kamayachi 1985; Engel and Last 2007). Security vulnerabilities are a special type of faults and may need special considerations for reliable estimation and prediction. It is more desirable to incorporate independent variables that have what are believed to be strong physical relationships

to security vulnerability, and not variables that are more generic in nature.

Alzhami et al. (O. Alhazmi, Malaiya, and Ray 2005; O. H. Alhazmi, Malaiya, and Ray 2007) have developed regression models around the vulnerabilities of major software, such as different releases of Windows and Red Hat Linux. These models may be able to predict the rate at which vulnerabilities are discovered in general but cannot pin point the components that may be more vulnerable than others. Hence, these models provide little actionable information. As discussed earlier, previous research has empirically showed that dependencies amongst software components have good vulnerability-prediction capability (Neuhaus et al. 2007; Neuhaus and Zimmermann 2009). Because of previous success of dependency-based prediction, we will actively explore it further in this proposal as well.

CP-Miner is a tool for finding copy-paste bugs in source code. This project was eventually turned into the commercial system Pattern Insight, albeit with some algorithmic changes and improvements. CP-Miner looks only for copy-pasted code similarities in source code. The tool functions by using tokenization in what the authors call a "code-based" system. This differs from other systems, including our own, that may use parse trees (syntax based), string-based systems, and semantic systems. Unlike CP-Miner, our project relies on semantic understanding of the code in addition to syntactic analysis. The authors of the paper discussing CP-Miner add that it is very difficult to find copy-paste bugs using static and dynamic analysis, mostly because the same exploits must be rediscovered on each incarnation. Theoretically, our system would solve this problem, however the time complexity would be much greater than for the same problem using CP-Miner. CP-Miner allows for quick discovery and comparison of all copy-paste errors. CP-Miner works by using an algorithm called subsequence mining. This algorithm maps similar-looking code to the same index value, in lines or in blocks. Then, it looks for groupings in other parts of the code that are similar to previously indexed blocks. Finally, it compares these segments, and sees if variable changes and bugs have propagated across the two blocks by using a "change" ratio. By comparing this ratio to a threshold, bugs are identified. Like the next system, ReDeBug, this is a statistical analysis, and very different in usage and implementation than our proposal.

Pattern Insight, an industry tool based on CP-Miner with algorithmic improvements, works approximately in the same way. However, the system relies on a larger database of comparison information, much like the DejaVu system which will be discussed later.

ReDeBug is a similar system to CP-Miner, but has some novel ideas. ReDeBug looks for "code clones", which by the authors' definition, are similar copies of code blocks across source files. In this way, it is much like CP-Miner, however, ReDeBug is a syntax-based system. Once again, the lack of semantic understanding, and the lack of expansion beyond copy-pasted code makes this system limited in a way that our project is not. ReDeBug operates by removing whitespace, special characters (like braces), and tokenizing source code lines. A sample of n tokens is taken, and compared against subsequent samples in a source file. Another parameter, theta, is used as a required threshold for similarity between code blocks. The combination of these parameters is used such that ReDeBug is a statistical analysis, rather than lexicographical analysis. To save on space and complexity constraints, the comparison vectors are stored in Bloom filters, which are then compared via unions. This analysis is again very different from our proposal, as semantic understanding of code is not achieved. In addition, the only discussion of security vulnerabilities found by ReDeBug, rather than non-security related bugs, uses the same clone-detection mechanism, so no additional complexity or analysis is added for security holes.

DejaVu is another tool for clone analysis, and acts basically as a superset of CP-Miner (in terms of it effectiveness in detecting bugs). It goes without saying the DejaVu is also dissimilar to our proposal, since once again semantic understanding is not achieved. DejaVu, while detecting a superset of CP-Miner bugs, is implemented in a very similar fashion to ReDeBug. The system forms an abstract syntax tree (AST), and uses an unstructured sequence of tokens (usually 30 to 50 tokens in length) in a sliding window on the AST to look for code clones. DejaVu applies a filtering technique using minimal textual similarity to cut down on the number of detected bugs, and eliminate benign detections. Like CP-Miner, the end goal of DejaVu is to find inconsistent changes across code clones. However, as can be seen by the use of the AST and a sliding window of tokens, the analysis performed is purely semantic, and, while in-depth, fails to reach the semantic understanding of our proposed use of PDGs and deep code analysis.

The last of the major systems is DECKARD. In a recurring theme, DECKARD is a clone detection system, using a novel algorithm for clone detection in an AST. Again, semantic analysis is not achieved, and the authors specifically point out that program data graph (PDG) construction is avoided in this system. The authors also mention the use of PDGs in other systems, but upon reviewing these references, we could find no uses of PDGs in a security- or bug-discovery-related context. The implementation details of DECKARD require extensive knowledge of complex mathematical formulas, which is beyond the scope of this analysis. However, it suffices to say that DECKARD, while complex, still does not take the semantic and deep-level understanding of code that our proposed project will.

The remaining systems, as aforementioned, are CCFinder, MOSS, and SYDIT, which are either referenced by the previous papers, found by independent research, or both. Nothing new is added by these systems, so we can safely say our project proposal is unique to the security analysis field (although the possibility of a similar system exists, we find it extremely unlikely). Without expounding extreme detail, we now summarize these systems.

CCFinder uses a tokenization and sliding window technique very close to ReDeBug and DECKARD. In fact, this system is so similar to ReDeBug that it could be seen as a spiritual clone.

MOSS is a plagiarism-detection tool used by Stanford Uni-

versity and others to detect copied code between student submissions. No implementation details could be found regarding this system, likely due to its purpose. However, evidence suggests that MOSS uses syntax-based clone detection, much like many of the aforementioned systems, and thus adds no novelty to our research.

Finally SYDIT is a system very close to DECKARD, and in fact, mentioned by the authors of the latter. It utilizes AST construction and contextual awareness to look for code "transformations", as coined by the authors. These transformations are spiritually the same as code clones, albeit with a different nomenclature.

This concludes our survey of previous and related work in the context of code analysis for security flaws. As evidenced by our research, our system would provide a unique and novel approach to source code analysis, and, while other systems have similar goals and share similar implementation mechanisms, none match the deep, semantic analysis of code that we wish to achieve. In addition, the use of PDGs seems fairly rare in current systems, which sets our system apart from others. Other systems use ASTs and database mining for code analysis, which we also plan to implement, but our combination of these two mechanisms, and the novel PDG construction make our system stand apart from others.

Not too much related work can be found about dependency analysis. Neauhaus (2007) examined the codebase of Mozilla and used both function calls and imported libraries as features to classify the vulnerable components. Neauhaus (2009) is an extension of their previous work, they used the similar machine learning techniques and apply it to the analysis of vulnerable software packages. Most of the work of this project is borrowing the idea of Neauhaus (2007).

5. PRESENTATION COMMENTS

During our final presentation, it was brought up that PDG comparison is a very difficult problem. Indeed this is true, and is a major motivation for the move from C++ to C code. As evidenced by our future work section, this is a continuing problem that requires additional work. Again, given the short time period we had for implementation, we were unable to produce a high-quality comparator. However, although difficult, we believe this comparison is possible , with a high degree of accuracy, given more effort.

6. CONCLUSION

In this paper, we have presented our system for predicting software vulnerabilities using a combination of PDG construction, dependency analysis, and comparison to mined vulnerability databases. We have exposed components for each of these features in turn, showing how each is implemented and what purpose it fulfills for the system as a whole. We have also explored how other researchers have attempted similar analyses, but have failed to produce the level of semantic understanding and code analysis that we have aimed for. Finally, we have discussed the shortcomings of our system, what remains to be implemented, and how these issues can be overcome. In short, we have provided the basis for an automated, semantic-level vulnerability detector, which leverages existing code bases to predict vulnerabilities in new and untested software. It is our hope that our work can be extended, as per our recommendations, into a functional vulnerability analysis tool.

7. REFERENCES

1 CCFinderX.
http://www.ccfinder.net/doc/10.2/en/whats.html.

2 Jiyong Jang, Abeer Agrawal, and David Brumley. "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions."

3 LIBSVM, http://www.csie.ntu.edu.tw/ cjlin/libsvm/

4 Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. "DECKARD: Scalable and Accurate Tree-based Detection of Code Clones."

5 Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2010.

6 "Moss: A System for Detecting Software Plagiarism."
http://theory.stanford.edu/ aiken/moss/

7 Na Meng, Miryung Kim, and Kathryn S. McKinley. "Sydit: Creating and Applying a Program Transformation from an Example."

8 Neuhaus, Stephan, and Thomas Zimmermann. 2009. "The beauty and the beast: vulnerabilities in red hat's packages."

9 Neuhaus, Stephan, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. "Predicting vulnerable software components." In Proceedings of the 14th ACM conference on Computer and communications security, 529-540.

10 Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou. "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code."
http://theory.stanford.edu/ aiken/moss/

APPENDIX

A. README

A.1 Code Location

https://github.com/mosqutip/EECS395-27

A.2 How to Run the Code

Frama-C: frama-c/frama-c -pdg-print -pdg -cpp-command 'gcc -C -E <options>' <file1.c> <file2.c> ...
parser/comparator: ./comparator.py <input_pdg_file>