Clonewise – Automatically Detecting Package Clones and Inferring Security Vulnerabilities

Silvio Cesare, Deakin University

Abstract— Developers sometimes statically link libraries from other projects, maintain an internal copy of other software or fork development of an existing project. This practice can lead to software vulnerabilities when the embedded code is not kept up to date with upstream sources. As a result, manual techniques have been applied by Linux vendors to track embedded code and identify vulnerabilities. We at Deakin University propose an automated solution to identify embedded packages, which we call package clones, without any prior knowledge of these relationships. We then correlate clones with vulnerability information to identify outstanding security issues. Our approach identifies similar source files based on file names and content to identify relationships between packages. We extract these and other features to perform statistical classification using machine learning. We evaluated our automated system against Debian's manually created database. Clonewise had a 58% true positive rate and a false positive rate of 0.03%. Additionally, our system detected many package clones not previously known or tracked. Our results are now starting to be used by Linux vendors such as Debian and to track embedded packages. Red Hat started to track clones in a new wiki, and Debian are planning to integrate Clonewise into the operating procedures used by their security team. Based on our work, over 30 unknown package clone vulnerabilities have been identified and patched.

Index Terms—Vulnerability detection, code clone, Linux.

1 Introduction

Developers of software sometimes embed code from other projects. They statically link against an external library, maintain an internal copy of an external library's source code, or fork the development of an external library. A canonical example is the zlib compression library which is embedded in much software due to its functionality and permissive software license. In general, embedding software is considered a bad development practice, but the reasons for doing so include reducing external dependencies for installation, or modifying functionality of an external library. The practice of embedding code is generally ill advised because it has implications on software maintenance and software security. It is a security problem because at least two versions of the same software exist when it is embedded in another package. Therefore, bug fixes and security patches must be integrated for each specific instance instead of being applied once to a system wide library. Because of these issues, for most Linux vendors, package policies exist that oppose the embedding of code, unless specific exceptions are required.

In the example of zlib, each time a vulnerability was discovered in the original upstream source, all embedded copies required patching. However, uncertainty existed in Linux distributions of which packages were embedding zlib and which packages required patching. In 2005, after a zlib [1] vulnerability was reported, Debian Linux [2] made a specific project to perform binary signature scans against packages in the repository to find vulnerable versions of the embedded library. To create a signature the source code of zlib was manually inspected to find a version string that uniquely identified it. This manual approach still finds vulnerable embedded versions of software today. We constructed signatures for vulnerable versions of compression and image processing libraries including bzip2, libtiff, and libpng. The version strings are shown in Fig. 1. We performed a scan of the Debian and Fedora Linux [3] package repository and found 5 packages with previously unknown vulnerabilities. Even for actively developed projects such as the Mozilla Firefox web browser, we saw windows of exploitability between upstream security fixes and the correction of embedded copies of the image processing libraries. Even in mainstream applications such as Firefox, these windows of opportunity sometimes extended for periods of over 3 months.

1.1 Motivation for Automated Approaches

The approach of manual searching for embedded copies of specific libraries deals poorly with the scale of the problem. According to the list of tracked embedded packages in Debian Linux, there are over 420 packages which are embedded in other software in the repository. This list was created manually and our results show that it is incomplete. Other Linux vendors were not even tracking embedded copies before our research supplied them with relevant data. It is evident from this that an automated approach is needed for identifying embedded packages without prior knowledge of which packages to search for. This would aid security teams in performing audits on new vulnerabilities in upstream sources.

Our approach is to consider code reuse detection as a binary classification problem between two packages. The classification problem is 'do these two packages share code?' We achieve this by performing feature extraction from the two packages and then performing statistical classification using a vector space model. The features we use are based on the filenames, hashes, and fuzzy content of files within the source packages

To identify security vulnerabilities we associate vulnerability information from public CVE advisories to vulnerable packages and vulnerable source files. We then discover all clones of these packages in a Linux distribution. Finally, we check the manually tracked vulnerable packages that Debian Linux maintain for each CVE and report if any of our discovered clones are not identified as being vulnerable.

2 Statistical Classification

Clonewise is based on machine learning. We employ statistical classification to learn and then classify two packages as sharing or not sharing code.

The classification problem is to have a known set of classes and then given an object, assign a class to it. Binary classification is when there are two classes. For example, spam and not spam in spam classification. Another example is malicious and non malicious in malware classification.

In general, classification can also be divided into supervised and unsupervised learning. In supervised learning, the classifier is trained using labeled data. The labeled data has objects with their classes. After training, objects with unknown classes can be classified. In unsupervised learning, there is no labelled data. The typical approach is then to use clustering to identify classes. Clustering groups similar objects into the same cluster.

Many classification algorithms work on feature vectors. A feature vector contains a fixed number of elements or dimensions. Each dimension represents a feature. The value of element might be the number of times that feature occurs, or it may a numeric value representing the feature directly. Features do not always need to be numeric. However, for the classification algorithms we use in Clonewise, numeric features are used.

Classification is a well studied problem in machine learning and software is available to make analyses easy. Weka is a popular data mining toolkit using machine learning that Clonewise uses to perform machine learning.

2.1 Feature Extraction

Feature extraction is necessary to perform classification. We need to select features that reflect if two packages share or do not share code. The list of features we use is shown in Fig. 1 and is discussed in the following subsections.

Number of Filenames

Our first set of features is simply the number of filenames in the source trees of the two packages being compared.

Source Filenames and Data Filenames

In Clonewise, we distinguish between two types of filename features. Filenames that represent program source code and programs that represent non program source code. We distinguish these two types of filenames by their file extension. The list of extensions used to identify source code is shown in Fig. 1. Almost all of the features in Clonewise are applied for both source and data filenames.

```
1.        N_Filenames_A
2.        N_Filenames_Source_A
3.        N_Filenames_B
4.        N_Filenames_Source_B
5.        N_Common_Filenames
6.        N_Common_Similar_Filenames
7.        N_Common_FilenameHashes
8.        N_Common_FilenameHash80
9.        N_Common_ExactFilenameHash
10.       N_Score_of_Common_Filename
11.       N_Score_of_Common_Similar_Filename
12.       N_Score_of_Common_FilenameHash
13.       N_Score_of_Common_FilenameHash80
14.       N_Score_of_Common_ExactFilenameHash80
15.       N_Data_Common_Filenames
16.       N_Data_Common_Similar_Filenames
17.       N_Data_Common_FilenameHashes
18.       N_Data_Common_FilenameHash80
19.       N_Data_Common_ExactFilenameHash
20.       N_Data_Score_of_Common_Filename
21.       N_Data_Score_of_Common_Similar_Filename
22.       N_Data_Score_of_Common_FilenameHash
23.       N_Data_Score_of_Common_FilenameHash80
24.       N_Data_Score_of_Common_ExactFilenameHash80
25.       N_Common_ExactHash
26.       N_Common_DataExactHash
```

Figure 1. Clonewise features.

Number of Common Filenames

To identify a relationship exists between two packages such that they share common code we use common filenames in their source packages as a feature. Filenames tends to remain somewhat constant between minor version revisions, and many filenames remain present even from the initial release of that software. For our purposes we can ignore directory structure and consider the package as a set of files, or we can include directory structure and consider the package as a tree of files. We noted several things while experimenting with this feature:

1. Many files in a package do not contribute to the actual program code.
2. C code is sometimes repackaged as C++ code when cloned. For example, lib3ds.c might become lib3ds.cxx.
3. The filenames of small libraries can often be referred to as libfoo.xx or foo.xx in cloned form.
4. Some files that are cloned may include the version number. For example, libfoo.c might become libfoo43.c.

We therefore employ a normalization process on the filenames to make this feature counting the number of similar filenames more effective.

Normalization works by changing the case of each filename to be all lower case. If the filename is prefixed with lib, it is removed from the filename. The file exten-

C	cpp	cxx	cc	php
inc	java	py	rb	js
pl	pm	m	mli	lua

Figure 2. Source code filename extensions.

sions .cxx, .cpp, .cc are replaced with the extension .c. Any hyphens, underscores, numbers, or dots excluding the file extension component are removed.

Number of Similar Filenames

It is useful to identify similar filenames since they may refer to nearly identical source code. A fuzzy string similarity function is used that matches if the two filenames are 85% or more similar in relation to their edit distance.

We chose the edit distance as our string metric after experimenting with other metrics including the smith-waterman local sequence alignment algorithm and the longest common subsequence string metric.

Number of Files with Identical Content

We perform hashing of file content using the ssdeep software and do a comparison of hashes between packages to identify identical content without respect to the filenames used. Like the previous class of feature, we have a feature for the number of files having identical content that are all program source code, and a feature for the number of files having identical content that are non program source code.

Number of Files with Common Filenames and Similar Content

To increase the precision of file matching from the previous feature, we employ a fuzzy hash of the file contents and then perform an approximate comparison of those hashes for files with similar filenames. While the previous approach is based on file names alone, the new approach is a combination of file names and content. Fuzzy hashing can be used to identify near identical data based on sequences within the data that remain constant using context triggered piecewise hashing [4]. The result of fuzzy hashing file content is a string signature known as its fuzzy hash. Approximate matching between hashes is performed using the string edit distance known as the Levenshtein distance. The distance is then transformed to a similarity measure. The similarity is a number between 0 and 100 indicates the hashes are not at all similar, and 100 indicates that the hashes are equal.

We have features for the number of files of similar content with a similarity greater than 0 of program source code and non program source code. We also count the number of similar files having a similarity greater than 80.

Scoring Filenames

Not all filenames should be considered equal. Filenames, such as README or Makefile that frequently occur in different packages should have a lower importance than those filenames which are very specific to a package such as libpng.h. We account for this by assigning a weight for each filename based on its inverse document frequency. The inverse document frequency lowers the weight of a term the more times it appears in a corpus and is often used in the field of information retrieval.

We use features scoring the sum of matching filename weights to the number of similar files, the number of similar files and similar content with similarity greater than 0 and 80, for both program source code and non program source code.

Matching Filenames Between Packages

If filename matching in VIII.A was performed as an exact match, then the number of filenames shared would be the cardinality of the intersection between the two sets of filenames. However, in Clonewise the filename matching is approximate based on the string edit distance. This means that some filenames such as Makefile.ca could potentially match the filenames Makefile.cba and Makefile.cb. Moreover, the scores for each filename as discussed in VIII.D can be different depending on which filename is deemed to be a match. We solve this problem by employing an algorithm from combinatorial optimization known as the assignment problem.

The assignment problem is to construct a one-to-one mapping between two sets, where each possible mapping has a cost associated with it, such that the mappings are chosen so that the sum of costs is optimal.

In our work the sets are the two packages and the elements of each set are the filenames in that package. The cost of the mapping between sets is the score of the matching filename in the second set according to its inverse document frequency. Our use of the assignment problem seeks to maximize the sum of costs.

The assignment problem can be solved in cubic time in relation to the cardinality of the sets using the Hungarian or Munkres [5] algorithm.

The Munkres algorithm is effective, however for large N, a cubic running time is not practical. We employ a greedy solution that is not optimal but is more efficient when N is large.

2.2 Classification

Clonewise uses the binary classification 'Do these two packages share code?' The two classes are 'shares code', and 'does not share code'. Clonewise is given a package name as input, and the classification problem is applied between that package and every other package in the repository. The output of Clonewise is the set of packages where the classification determines the package pairs share code. Clonewise also reports the filenames between the packages and the weights of those filenames.

Clonewise uses supervised learning to build a classification model. We use the manually created Debian embedded-code-copies database that tracks package clones to train and evaluate our system.

3 Scaling The Analysis

Our system is effective and reasonably efficient at identifying clones in a single Linux package. However, in a typical Linux distribution there exist more than ten thousand individual packages. Our system would be impractically long if we performed clone detection on all packages without taking advantage of multicore and cluster computing.

Our implementation employs both shared and distributed memory models. Therefore, we classify it as a hybrid model. It takes advantage of both multicore for shared memory, and cluster computing for distributed memory.

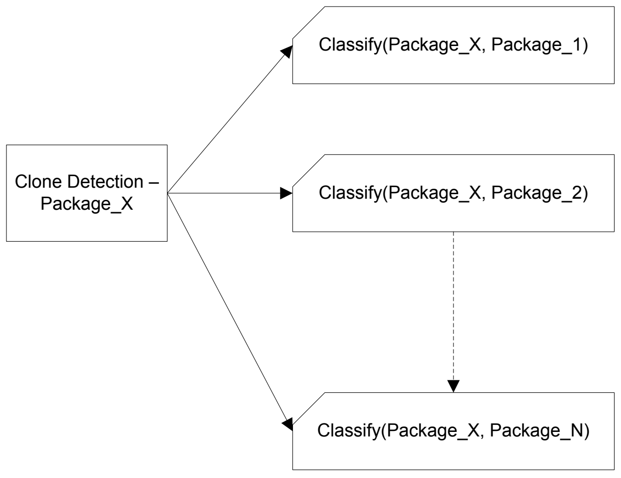
Clone Detection – Package_X
Classify(Package_X, Package_1)
Classify(Package_X, Package_2)
Classify(Package_X, Package_N)

Figure 3. Multicore.

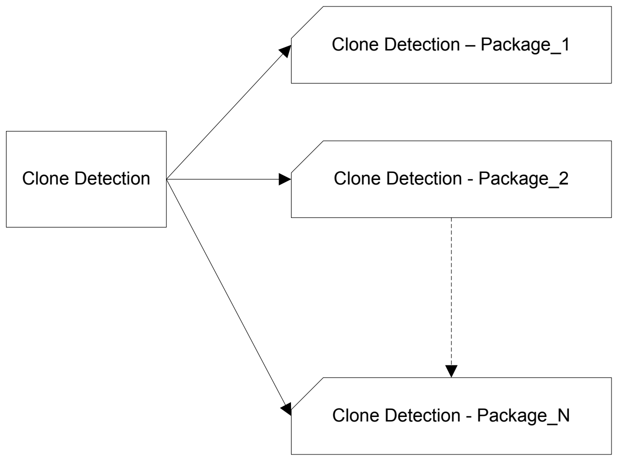
Clone Detection
Clone Detection – Package_1
Clone Detection - Package_2
Clone Detection - Package_N

Figure 4. Clustering.

3.1 Multicore

Given an input package to perform clone detection, Clonewise pairs that package with every other package in a Linux distribution. These package pairs are the input to a binary classification problem. Each binary classification problem can be evaluated independently of the other binary classification problems. This model of evaluation is embarrassingly parallel and leads to efficient parallel and distributed computing as shown in Fig. 3.

We chose to solve this problem using multicore computing. We used the Open MP multicore programming model to implement our solution. Open MP is a shared memory model based on the use of compiler directives. We parallelize the feature extraction and classification for each package pair. This process improves the speed it takes to perform clone detection on an individual package.

3.2 Clustering

Our multicore implementation improves the performance of clone detection on a single package. We use cluster computing to distribute clone detection of multiple packages. Each package can be scanned in parallel without regard to other packages and is also an embarrassingly parallel problem leading to efficient parallel and distributed implementations as shown in Fig. 4.

We implemented our system using message passing with Open MPI. In our implementation, a job is defined as performing clone detection on a single package. Since we have many packages to analyse, a master node distributes jobs to slave nodes. When the slaves complete a job they signal the master node requesting more work.

3.3 Running the Analysis

We analysed our Linux distribution using a high performance computing compute cluster. We purchased 4 hours of cluster computing time from the Amazon EC2 cloud computing service. We built a 4 node cluster with dual CPUs per node, Intel Xeon E5-2670, eight-core "Sandy Bridge" architecture), 60.5G of memory per node, and CPU performance identified as 88 EC2 compute units.

In our initial implementation we used the MPI_Scatter API call to statically distribute jobs to nodes. We found that due to the variety in times to complete jobs, it performed poorly because some nodes quickly finished their work and then idled, while other nodes continued processing.

We also realized that it was important to use multicore when possible to increase the speed of individual jobs. We tried initially to favour clustering over multicore and we found that a few jobs would take significantly longer to complete and were the bottleneck for completing analysis of our dataset. Multicore makes these jobs complete faster and leads to better utilization of cores in our cluster. Another advantage of using shared memory multicore is that the memory usage is lower since fewer processes are running on each node. This was not a problem for us on our Amazon EC2 cluster as each node had 60.5G of memory. However, on nodes with less memory, it could become a significant issue.

4 Inferring Security Vulnerabilities

In this section, we propose use-cases for clone detection by Linux security teams. We also propose a completely automated solution to find out-of-date clones that have outstanding security vulnerabilities.

4.1 Use-case of Clone Detection to Detect Vulnerabilities

One method which we initially tried was to look at packages that had reported vulnerabilities against them. We considered this list as security sensitive packages. We used this list of packages as input to our clone detection analysis. Anytime a security sensitive package was cloned, we verified that the clone was not out of date. This is an effective method to detect vulnerabilities, but it requires manual analysis.

Even though the technique we described is manual, it still has benefits today and can be used in an on-going basis to detect new vulnerabilities.

If a new vulnerability is found in a package, then clone

Summary: Off-by-one error in the __opiereadrec function in **readrec.c** in libopie in OPIE 2.4.1-test1 and earlier, as used on FreeBSD 6.4 through 8.1-PRERELEASE and other platforms, allows remote attackers to cause a denial of service (daemon crash) or possibly execute arbitrary code via a long username, as demonstrated by a long USER command to the FreeBSD 8.0 ftpd.

Figure 5. An NVD CVE summary.

detection should be performed on the Linux distributions because it is likely the same vulnerability is present in the cloned software. For example, if a vulnerability is reported for libpng, then clone detection should be performed and each libpng clone checked to see if the vulnerability is present. This method can be used by Linux security teams, but for old vulnerabilities it is not advisable since many clones would be patched but not reported by a Linux vendor. Therefore, we looked at other automated methods to detect out-of-date clones which we describe in the following sub-sections.

4.2 Debian Linux Security Tracking

Debian Linux make a significant effort to track security information and maintain a publicly accessible repository known as the security tracker for tracking security problems in their distribution.

A useful database that is unique to Debian is a manually generated list is used to associate CPE names to Debian package names (CPE is a package name standard across different vendors). This is done so Debian can check native packages against new vulnerabilities that appear as a CVE in the NVD (CVE is a vulnerability reporting standard).

Debian Linux also use CVE internally to track vulnerabilities. They maintain a database of every CVE. They then list every package in Debian affected by each particular CVE.

4.3 Automated Vulnerability Inference

In Clonewise, we can use clone detection in addition to the above information to identify untracked vulnerabilities.

1. Clonewise takes a CVE number as input and extracts the vulnerable package from the report. The CPE package name is translated to a native Debian package name.
2. Clonewise then parses the summary to find the vulnerable source files. It is possible to extract theses vulnerable source files from the summary (Fig. 5) by tokenizing the summary into words and extracting words that have a file extension of known programming languages.
3. Clonewise then looks at all the clones of the vulnerable package and trims the list by ensuring one of the vulnerable source files is present in the clone and that the fuzzy hash between the vulnerable package's source is similar to the clone's.
4. We also trim the list by ignoring clones that we believe have been patched to use the system wide dynamic library. We did this by checking if in the binary version of the package the embedded package was a package dependency. If the embedded package is a dependency, then it the main package almost certainly uses it for dynamic linking. Dynamic linking is the normal approach vendors use to address the security implications of package clones.
5. Finally, Clonewise checks to see if Debian Linux is tracking this package clone as being affected by that particular CVE. If it is not being tracked, then Clonewise will report the package as being potentially vulnerable.

This process of finding outstanding vulnerabilities is applied to every CVE of interest in the database, and a final report is generated. The normal process is that a security analyst then verifies each reported vulnerability and eliminates any false positives.

One feature that we didn't implement was using the CVE summary's reference to vulnerable functions. We could potentially parse the sentence containing the vulnerable source filename to extract the vulnerable function and then check for the presence of this string in the source file. We did not do this because it requires the contents of each source tree to be maintained as signatures. This would increase the data storage requirements of Clonewise which we thought to be impractical. Potentially we could download the source as required, but this would cause issues doing analysis between distributions.

5 Implementation and Evaluation

In this section we discuss our Clonewise implementation and the evaluation of our implementation.

5.1 Implementation

We implemented a complete system named Clonewise to identify package clones in Linux distributions. Clonewise automatically downloads a Linux distribution package repository and builds a database of signatures for each package. It then trains a model and uses statistical classification to perform clone detection. We employ the Weka machine learning toolkit to perform the data mining aspects of our system.
Our implementation uses C++ and shell scripting. It consists of about 3,500 lines of code (LOC) to perform the package clone detection and vulnerability inference.

We performed an analysis of the Ubuntu Linux distribution and also performed some analysis of other distributions including Fedora 13 and Debian Linux. The package count in each distribution was in excess of 10,000.

Clonewise consists of multiple components. The components are divided into:

1. Parsing Debian's package clone database
2. Building the Clonewise database
3. Training the classification model
4. Clone detection
5. Building a clone detection cache
6. Querying the cache
7. Finding cloned files
8. Inferring vulnerabilities

TABLE 1. UNKNOWN FEDORA LINUX VULNERABILITIES

Package	Embedded Package
OpenSceneGraph	lib3ds
mrpt-opengl	lib3ds
mingw32-OpenSceneGraph	lib3ds
libtlen	expat
centerim	expat
mcabber	expat
udunits2	expat
libnodeupdown-backend-ganglia	expat
libwmf	gd
Kadu	mimetex
cgit	git
tkimg	libpng
tkimg	libtiff
ser	php-Smarty
pgpoolAdmin	php-Smarty
sepostgresql	postgresql

TABLE 2. PREVIOUSLY UNKNOWN DEBIAN LINUX VULNERABILITIES

Package	Embedded Package
boson	lib3ds
libopenscenegraph7	lib3ds
libfreeimage	libpng
libfreeimage	libtiff
libfreeimage	openexr
r-base-core	libbz2
r-base-core-ra	libbz2
lsb-rpm	libbz2
criticalmass	libcurl
albert	expat
mcabber	expat
centerim	expat
wengophone	gaim
libpam-opie	libopie
pysol-sound-server	libmikod
gnome-xcf-thumnailer	xcftool
plt-scheme	libgd

We parse Debian's package clone database and convert it to XML or a text based format. We can optionally filter the results to ignore statically linked clones, or we can filter those clones which have been fixed, or those clones which remain unfixed. This component is necessary for generating the labelled training data to build a machine learnt model for classification.

We can also find clones of files given as input. The output is the set of packages that have a similar file in their source trees.

TABLE 3. ACCURACY OF STATISTICAL CLASSIFICATION

Classifier	TP/FN	FP/TN	TP Rate	FP Rate
Naïve Bayes	439/322	484/56296	57.69%	0.85%
Multilayer Perceptron	204/557	48/56732	26.81%	0.08%
C4.5	523/238	86/56694	68.73%	0.15%
Random Forest	533/228	60/56720	70.04%	0.11%
Random Forest (0.8)	446/315	15/56765	58.61%	0.03%

To build the Clonewise database, we download the entire source package repository for a Linux distribution, unpack the sources, and generate signatures. The signatures are the ssdeep signatures of the source trees for each package. We also build a package index relating binary packages to source packages. Finally we build a package dependency list for the purpose of identifying fixed clones.

Clone detection is performed as explained earlier by using machine learning. XML output is optional.

The clone detection cache is built using a cluster and the results of clone detection are stored to disk.

The cache can be queried so that clone detection does not need to be performed again. XML output is optional.

Finally, vulnerability inference relates clones in the cache to Debian's security tracker and the NVD CVE information.

5.2 Establishing Ground Truth

Debian Linux maintain a manually created database of packages that are cloned in their security tracker. This database was not originally created to be processed by a machine, so some of the data is not consistent in referencing packages with their correct machine readable names. Instead, shorthand or common names for packages and libraries are sometimes used. We cull all those entries which do not reference package sources and are therefore not suitable for our system.

To establish true negatives, we randomly selected pairs of packages not in our true positive list. We label these package pairs as negatives. This data can be unclean since we observe the labeled true positives are incomplete, but even so, the true negatives we label are still useful for training our statistical model. In total, we obtained 761 labelled positives and 56780 negatives.

5.3 Accuracy of Statistical Classification

We experimented with a number of classification algorithms and employed 10-fold validation from our labeled dataset to evaluate the accuracy of our system. Our results are shown in Table 3. We obtained the best result using the Random Forest classification algorithm. This classification algorithm performed significantly better than all other algorithms we evaluated. The true positive rate is 70.04% which we think is quite reasonable for the first implementation of an automated system for package clone detection. The false positive rate must be very low for our system to be used by Linux security teams. Our initial false positive rate is 0.11%. We then modified the decision threshold of the random forest algorithm to consider false positives as more significant than false negatives. Our false negative rate is 0.03% with a decision threshold of 0.8 which represents that 3 in every 10,000 package pairs is mislabeled as a positive. The true positive rate is lower with a higher decision threshold and is 58.61%. This is the trade-off we accept for a low false positive rate. There are about 18,000 source packages, so there are 18,000 package pairs that are classified when performing clone detection on an individual package. Therefore, if our training data were not noisy, we would predict 4 to 5 false positive per complete clone detection on an individual package. However, our labelled negatives are noisy, and some negatives are actually positives. Therefore, we think between 4 to 5 false positives is closer to an upper limit. This is reasonable for a manual analyst to verify and we think it will not cause significant burden on Linux security teams.

5.4 Package Clone Detection

As part of the practical results from our system we contributed 34 previously untracked package clones to Debian Linux's embedded code copies database. Thus, we feel that the package clone detection provides tangible benefit to the Linux community.

We also verified if the embedded packages we detected were not in fact patched by the Linux vendors to link dynamically against a system wide library.

5.4 Vulnerability Detection

The direct consequence of package clone detection is determining if a clone is out of date and if it has any outstanding and unpatched vulnerabilities. As part of our work we detected over 30 vulnerabilities in Debian and Fedora Linux because of package clone issues by checking security sensitive packages automatically, manually, or using adhoc identification of out-of-date clones. The vulnerabilities in each package we found using clone detection are shown in Table 1 and 2.

We performed a more recent evaluation of completely automated vulnerability inference over the years of 2010, 2011, and 2012. We found a number of new vulnerabilities and are currently working with vendors to remedy these issues. The results of our system demonstrate that we effectively identify vulnerabilities with a false positive rate that is practical for manual verification.

6 Related Work

Large scale manual attempts at auditing specific Linux distributions for embedded packages have occasionally occurred in the past. In 2005, the Debian package repository was scanned for vulnerable zlib fingerprints based on version strings [6]. Antivirus signatures were generated and ClamAV performed the scanning. Our system improves practice by automating the discovery of embedded packages without prior knowledge of which packages are embedded. Additionally, our system automatically constructs the signatures to detect embedded packages.

Software similarity is an area that covers topics such as malware variant detection, software theft detection, plagiarism detection, and code clone detection. This area identifies similarities between software and may be applied to binary or source code. Our work is a similar problem in that of finding similar occurrences of packages embedded in other packages. The most related works to ours is that of software clone detection [7] and plagiarism detection because they are a source level software similarity problems. Clone detection identifies duplicated copies of code fragments. This can be used to identify duplication of effort in source code which can be a source of software bugs or confusion. Plagiarism also detects source code that has been copied between software. Our system is not as fine grained as code clone or plagiarism detection and detects code similarity at the source file and package level. This allows us to use integrate our system into existing practice by Linux vendors, and allows us to use vulnerability information which is provided at the package level. We believe that while our approach is simplistic, this method offers practical and immediately useful benefits to practitioners.

6.1 Plagiarism Detection

Plagiarism detection systems often make the distinction between attribute counting and structure based techniques. Attribute counting is based on software metrics, or the frequencies of particular features occurring. Typical approaches include Halstead metrics and other metrics which take into account attributes including the number of tokens, the number of operators, the number of variables, or the number of source lines [8]. Structure based techniques rely on using program structure which typically include the use of dependency graphs or parse trees.

JPlag [9] and YAP3 [10] consider tokens from source code as features and perform similarity comparisons using greedy string tiling. Another approach [11] considers tokenization and linearization of the source code and uses an adaptive sequence alignment to construct a similarity measure.

Parse trees are related to abstract syntax trees and have been proposed for plagiarism detection [12] by using tree comparisons to identify similarity. Tree similarity can be based on algorithms including tree edit distances or largest common subtrees.

GPLAG used program dependency graphs of programs [13]. Similarity between program dependency graphs uses similarity metrics such as the graph edit distances.

6.2 Code Clone Detection

Clone detection can be performed on the textual stream in a source file once whitespace and comments are removed [14]. The key concept is that a fingerprint of a code fragment is obtained and then the remainder of the source scanned for possible matching duplicates. More recently [15, 16] has used the token approach with good success in large scale evaluations. Large scale copy and paste clones using a data mining approach was investigated in [17, 18].

An alternative approach is to use the abstract syntax tree of the source to generate a fingerprint [19]. Tree matching can subsequently be used to discover software clones. Abstract syntax trees are more impervious to superficial changes to the textual stream and textual organization of the code.

Other program abstractions can be used to fingerprint code fragments such as the program dependency graph which is a graph combining control and data dependencies [20].

In non exact matching of code fragments, similarity searches can be used using appropriate distance metrics such as the Euclidean distance, given an appropriate threshold for similarity. Using non exact matching of code fragments allows detection of duplicated code that has been revised or that subjected to an evolutionary process. Our system allows for evolution and revision of code by using fuzzy hashing over the source.

7 Future Work and Conclusion

There are several ways we could continue our research or see it applied to improve current practice. We could apply our system to more source code, including other Linux distributions, BSD vendors and also online source code repositories such as Google Code [21] or Sourceforge [22]. It is conceivable that source code repositories could offer services to find package clones. Our system could be integrated into a package build system to automatically update the embedded database information or ask for validation from a package maintainer. Debian Linux would like our Clonewise tool to run constantly in the background and scan the source code repository to update a live database of clones. If we did this, we could enforce build recommendations that aim for avoidance of embedded code. Finally, the embedded package information could automatically be cross referenced against new advisories relating to embedded code. The Debian Linux security team has asked us to perform this integration into their distribution as part of a standard operating procedure for when a vulnerability is found in a package and this is a focus of our current work.
In addition to the number of reported vulnerabilities and subsequent patching and resolution of vulnerabilities, we believe our research has much value in the practical approach of coping with embedded code and packages that may or may not be known about. We believe all vendors benefit in creating and maintain databases of embedded code in their package repository and our research fills a gap when the manual task of auditing in excess of 10,000 packages per distribution is too time consuming to be practical. There is much work as a consequence that could be applied to current practice to aid package maintenance and cross checking of advisories relating to embedded code and feel our work is a large step towards this goal.

References

[1] J.-l. Gailly and M. Adler. (2011). *zlib*. Available: http://zlib.net

[2] (2011). *Debian Linux*. Available: http://www.debian.org

[3] Red_Hat. (2011). *Fedora Linux*. Available: http://fedoraproject.org

[4] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation,* vol. 3, pp. 91-97, 2006.

[5] H. Kuhn, W., "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly,* 1955.

[6] Christoph Biedl, Mark Adler, and F. Weimer. (2011). *Discovering copies of zlib*. Available: http://www.enyo.de/fw/security/zlib-fingerprint/

[7] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR,* vol. 541, p. 115, 2007.

[8] E. L. Jones, "Metrics based plagarism monitoring," *Journal of Computing Sciences in Colleges,* vol. 16, pp. 253-261, 2001.

[9] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science,* vol. 8, pp. 1016-1038, 2002.

[10] M. J. Wise, "YAP3: improved detection of similarities in computer program and other texts," *SIGCSE Bull.,* vol. 28, pp. 130-134, 1996.

[11] J.-H. Ji, G. Woo, and H.-G. Cho, "A source code linearization technique for detecting plagiarized programs," *SIGCSE Bull.,* vol. 39, pp. 73-77, 2007.

[12] J.-W. Son, S.-B. Park, and S.-Y. Park, "Program Plagiarism Detection Using Parse Tree Kernels," in *PRICAI 2006: Trends in Artificial Intelligence*. vol. 4099, Q. Yang and G. Webb, Eds., ed: Springer Berlin / Heidelberg, 2006, pp. 1000-1004.

[13] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," presented at the Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, Philadelphia, PA, USA, 2006.

[14] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," 1999, p. 109.

[15] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering,* pp. 654-670, 2002.

[16] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," in *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*, 2007, pp. 106-115.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI '04)*, 2004, pp. 20-20.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering,* pp. 176-192, 2006.

[19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," 1998, p. 368.

[20] J. Krinke, "Identifying similar code with program dependence graphs," 2001, p. 301.

[21] Google. (2011). *Google Code*. Available: http://code.google.com/

[22] Geeknet. (2011). *Sourceforge*. Available: http://sourceforge.net/