Clonewise – Detecting Package-Level Clones Using Machine Learning

Silvio Cesare, Yang Xiang, and Jun Zhang

School of Information Technology
Deakin University
Burwood, Victoria 3125, Australia
{scesare,yang,jun.zhang}@deakin.edu.au

Abstract. Developers sometimes maintain an internal copy of another 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. We propose an automated solution to identify clones of packages without any prior knowledge of these relationships. We then correlate clones with vulnerability information to identify outstanding security problems. This approach motivates software maintainers to avoid using cloned packages and link against system wide libraries. We propose over 30 novel features that enable us to use to use pattern classification to accurately identify package-level clones. To our knowledge, we are the first to consider clone detection as a classification problem. Our results show our system, Clonewise, compares well to manually tracked databases. Based on our work, over 30 unknown package clones and vulnerabilities have been identified and patched.

Keywords: 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 as a bad development practice, but the reasons for doing so include reducing external dependencies for installation, or the need to modify 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, in the past, uncertainty existed in Linux distributions of which packages were embedding zlib and

T. Zia et al. (Eds.): SecureComm 2013, LNICST 127, pp. 197–215, 2013.

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 and time consuming 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. 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.

The traditional approach for discovering duplicated fragments of insecure code has been through the use of code clone detection. Code clone detection applies pattern recognition on the syntactic or structural nature using the insecure code fragment as a template. Then a search is performed over other code to identify duplication or near identical duplication.

1.1 Motivation for Package-Level Clone Detection

Clone detection theoretically solves the problem of insecure code fragments propagating to other locations. However, in practice the number of code clones is significantly high. For developers of individual projects, clone information may be useful. Yet, package maintainers and operating system distributions have no realistic actions to take with such clone information since they are not the primary developers of the software they release. What package maintainers and operating system vendors want is the ability to repackage or build the software in such a way that improves security and eliminates clones. If vendors know that an entire package is cloned in another, then they can modify the build process to use the operating system's system wide library package. This is an achievable goal and improves the security and stability of the system. This is our motivation and the reason we see package-level clone detection as an important addition to software engineering that traditional clone detection does not address.

1.2 Motivation for Automated Approaches

The approach of manually 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. This identifies the motivation for our system named Clonewise to identify package-level clones.

Previous systems that automate and address part of the problem are software provenance systems. Our system extends such works by recognising more features in software that can be used to fingerprint pacakges. Our system also addresses the problem of software being implemented in multiple languages, even within the same package. Our work is language agnostic. We also address the problem of requiring every version of a software to match it against a query. Our system can determine if a package is embedded, irrespective of which version number is used. This has advantages, but also makes identifying security problems in specific versions harder. We overcome this by using side-information that tracks the necessary information and that is maintained by operating system vendors.

Our work is also similar to the concept of structural or higher-level clones as proposed in [4]. We are much more specific in the type of structure we are searching for. That is, package-level clones. The structural clones in [4] use directory-level clones to simulate module-level clones which is not as accurate.

1.3 Generality

At first glance, package-level clone detection may appear to be a Linux distribution specific problem. However, this problem applies to any vendor who maintains a repository of software packages and shares common code amongst packages. This problem also applies to any vendor which for legal reasons needs to know the provenance of embedded packages such as open source libraries. Finally, the problem applies to any vendor who needs to know what open source libraries have been embedded so as to keep up-to-date with upstream releases. It is quite conceivable that any large software project may incorporate some permissively licensed open source software as an embedded library or package. For all of these reasons, software engineering needs to incorporate automated means to provide assurance that the state of software and the existence of package-level clones is known.

1.4 Innovation

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 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 vulnerability and report if any of our discovered clones are not identified as being vulnerable.

In this paper we make the following contributions:

- We define the problem of package clone detection, and the sub-categories of shared and embedded package clone detection.
- We are the first ones to formulate code reuse detection as a pattern classification problem. Then, it is feasible to apply traditional pattern

classification algorithms to achieve accurate clone detection. We employ a novel asymmetric bagging based classifier combination method to address the specific classification problem.
- We propose over 30 new features for the purpose of clone detection, which are fundamental to solve the specific pattern classification problem. In particular, the proposed features are basis to the accuracy of clone detection.
- We propose applications of package clone detection. We present algorithms to identify outstanding security vulnerabilities based on out-of-date clones.
- We implement a complete system, Clonewise, which demonstrates our system effectively identifies package clones, finds vulnerabilities and is useful to vendors. For example, Debian Linux is planning infrastructure integration of Clonewise.

The structure of this chapter is as follows: Section 2 defines the problem of package clone detection and outlines our approach. Section 3 describes how Clonewise detects shared and embedded package clones using machine learning. Section 4 describes the algorithms we use to identify vulnerabilities based on clone information. Section 5 evaluates our system. Section 6 examines related work. Section 7 outlines future work. Finally we present our conclusions in Section 8.

2 Problem Definition and Our Approach

2.1 Problem Definition

A package clone is the duplication of one package's code in another package. It is the presence of code reuse between packages. How do we find these package clones?

A package can be embedded in another package. How do we determine this knowing that a package clone exists?

A package clone may contain vulnerabilities or other security problems because the clone is out of date. How do we find these?

2.2 Our Approach

Our approach for detecting clones is based on binary classification and shown in Fig. 1 and described below. A key point is that if two packages share code, one is not necessarily embedded in the other. We therefore detect code reuse and embedding as related but distinct problems.

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 perform 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.

A package clone consisting of two packages can be analysed to determine if one package is embedded in the other. We use a binary classification problem to answer this. The features we use are based on the size of the cloned code relative to the size of each package, and other features such has how many packages are dependent on the packages we are analysing.

We determine vulnerable packages by correlating security tracking information with our package clone detection analysis.

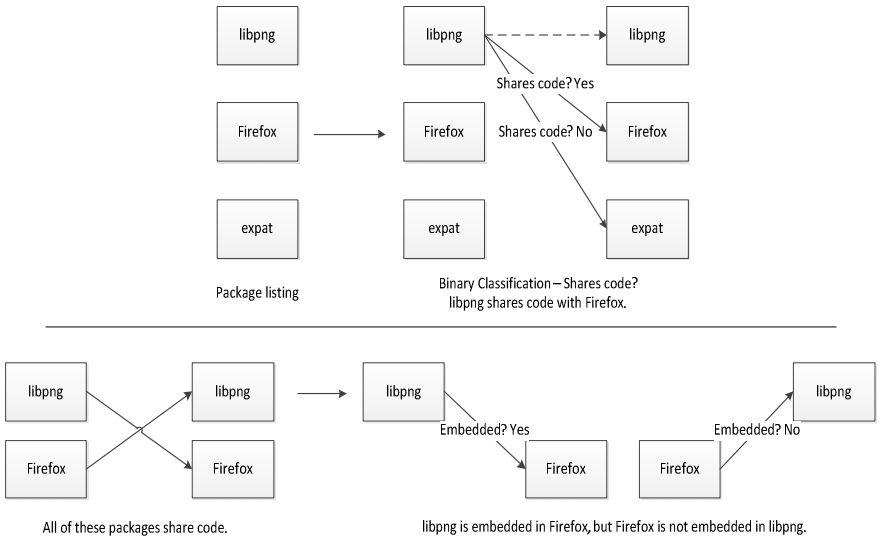

Fig. 1. Shared package clone detection (above) and embedded package clone detection (below)

3 Package Clone Detection

Clonewise is currently based on machine learning and we have found this approach to be most versatile and successful. We employ statistical classification to learn and then classify two packages as sharing or not sharing code.

Classification is a well-studied problem in machine learning and software is available to make analyses easy. Weka [5] is a popular data mining toolkit using machine learning that Clonewise uses to perform machine learning.

3.1 Shared Package Clone Detection

Feature extraction is necessary to perform shared package clone classification. We need to select features that reflect if two packages share or do not share code. The feature vector we extract is obtained from a pair of packages that we are testing for sharing of code. The 26 features we use are 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 are c, cpp, cxx, cc, php, inc, java, py, rb, js, pl, m, mli, and lua. Almost all of the features in Clonewise are applied for both source and data filenames.

Number of Common Filenames
To identify that 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: Many files in a package do not contribute to the actual program code. C code is sometimes repackaged as C++ code when cloned. For example, lib3ds.c might become lib3ds.cxx. The filenames of small libraries can often be referred to as libfoo.xx or foo.xx in cloned form. 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 extensions .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.

Our similarity measure is defined as:

$$similarity(s,t) = 1 - \frac{edit\_dist(s,t)}{\max(len(s), len(t))}$$

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 [6]. 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 measured is a number between 0 and 100. Zero indicates that 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 [7]. 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.

The inverse document frequency is defined as:

$$idf(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

where D is the set of packages, d is a package, and t is a filename in a package.

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 between two packages 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 the previous section 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 bijective 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. Formally, the assignment problem is defined as:

Given two sets, A and T, of equal size, together with a weight function C: A × T → R. Find a bijection f: A →T such that the cost function:

$$\sum_{a \in A} C(a, f(a))$$

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 [8] 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.

3.2 Shared Package Clone Classification

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. We employ a number of classifiers to evaluate our system as described in Section 7.

3.3 Embedded Package Clone Detection

To detect embedded package clones we use the results of shared package clone detection and apply a filtering stage to exclude packages where the first package is not embedded in the second package. We solve this problem by considering the problem as a binary classification problem.

Similar to the shared package clone detection approach, we perform feature extraction before using statistical classification. The 18 features we use are summarized in the following:

Number of Filenames
As in shared package clone detection, the number of filenames that are source and data are used.

Percent of X Embedded in Y
These features say how much of one package is embedded in the other package.

Package X has Lib in Name
These features are useful in identifying if a package is a library, which increases its likelihood that it is an embedding. If the package name is prefixed with ‘lib’, then the

feature is assigned a value of 1. If the prefix is not that, then the value is 0. The prefix is compared without regard to case.

A to B Ratio
These features inform us on how big the packages are relative to each other. It is typical that an embedded library is smaller than the software it is embedded in.

Package Dependents
These features inform us on how many other packages depend on the package in question. Libraries are typically used by many other packages and so the value for this feature will also be high. As explained earlier, that the package is library indicates that the package is more likely to be embedded.

3.4 Classification Using Asymmetric Bagging

For training our classifier, we have a finite set of labelled positive cases as obtained from vendor generated databases and we are able to arbitrarily generate labelled negative cases. We have many more negative cases than we have positive cases, wherein a positive case indicates an embedded package clone. This scenario represents the imbalanced class problem [9] where many classifiers favour the majority class. We decided to improve our detection rate of the positive class by addressing the imbalanced class problem by performing asymmetric bagging [10].

Asymmetric bagging uses all the labelled positive cases and use an equivalent number of negative cases obtained from a random sampling. This extends traditional bagging which uses a random and equal sampling from both classes. The asymmetric bagging approach described generates a single bag upon which a classification model is built from training. Many bags are created and classification models are built for each bag. When performing classification of an unlabelled instance, each bag makes a prediction and the results are aggregated using a majority vote. This has the effect or improving the accuracy when detecting positive cases. We implemented the asymmetric bagging algorithms by extending the bagging meta-classifier in the Weka machine learning toolkit.

4 Inferring Security Problems

In this section, we examine algorithms and approaches to detect software vulnerabilities. Package-level clone detection is not strictly the best method to discover security problems through code cloning. However, it is almost impossible in practice to apply code-level clone detection across tens of thousands of packages with potentially hundreds of thousands of clones and expect developers to integrate fixes. The reality is, a vendor's security team can fix high impact bugs and push for package maintainers to build their software using system wide package-level libraries. In effect, the only practically used system of bug fixing on a large scale in regards to clones, is by fixing package-level clones. Yet the problem still exists of how to motivate package maintainers or security teams to apply these fixes. The current

practice is to highlight that the cloned package contains known security problems and pointing out that there is less cost in rebuilding the software to eliminate the higher-level clone than it is to apply individual patches. Therefore, we see value in Clonewise as being a tool that can bring about good practices of eliminating package clones by highlighting vulnerabilities. To achieve the task of vulnerability detection, 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, for the purpose of vulnerability detection, was to look at packages that had reported vulnerabilities against them. We considered this a list of 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 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 Automated Vulnerability Inference

In Clonewise, we can use clone detection in addition to Debian Linux's security tracking information to identify untracked vulnerabilities.

Clonewise takes a vulnerability report given as a CVE (Common Vulnerabilities and Exposures) number as input and extracts the vulnerable package from the data. The standardized package name associated with the vulnerability, given as a CPE (Common Platform Enumeration) package name, is translated to a native Debian package name.

Clonewise then parses the summary of the CVE report to find the vulnerable source files. It is possible to extract theses vulnerable source files from the summary by tokenizing the summary into words and extracting words that have a file extension of known programming languages.

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.

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 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.

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.

5 Results and Evaluation

In this section, we discuss how we use an Amazon EC2 cluster to generate our results. We then discuss how our system performs against a labelled dataset of package clones and the security vulnerabilities our system has discovered. Finally, we discuss a web service to perform online scanning of software using our EC2 generated database.

5.1 Clonewise Compute Cluster

Our system employs multicore and clustering. We analysed our Linux distribution using a high performance 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. We then performed package-level clone detection on this infrastructure.

5.2 Establishing the Ground Truth for Training and Evaluation

Debian Linux maintain a manually created database of packages that are cloned in their security tracker database. We use this list of entries to establish the ground truth for our labelled data in an evaluation.

The Debian 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.

Table 1. Accuracy of Shared Package Clone Detection

CLASSIFIER	PRECISION	RECALL	ACCURACY	F-MEASURE
Naïve Bayes	0.47562	0.57687	0.98599	0.52137
Multi. Perceptron	0.80555	0.26806	0.98948	0.40225
C4.5	0.85878	0.68725	0.99436	0.76349
Random Forest	0.89881	0.70039	0.99499	0.78728
Rand. Forest (0.8)	0.96746	0.58607	0.99426	0.72994

Table 2. Accuracy of Shared Package Clone Detection

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%

Table 3. Accuracy of Embedded Package Clone Detection

CLASSIFIER	PRECISION	RECALL	ACCURACY	F-MEASURE
Naïve Bayes	0.10171	0.94349	0.35580	0.18362
Multi. Perceptron	0.75229	0.43101	0.94540	0.54802
C4.5	0.89235	0.75164	0.97396	0.81597
Random Forest	0.89067	0.72798	0.97225	0.80114
Asym. Bagging	0.53196	0.91852	0.93168	0.67372

Table 4. Accuracy of Embedded Package Clone Detection

CLASSIFIER	TP/FN	FP/TN	TP RATE	FP RATE
Naïve Bayes	718/43	6341/2808	94.35%	69.31%
Multilayer Perceptron	328/433	108/9041	43.10%	1.18%
C4.5	572/189	69/9080	75.16%	0.75%
Random Forest	554/207	68/9081	72.80%	0.74%
Asymmetric Bagging	699/62	615/8534	91.86%	6.72%

We had two types of negative labeled entries where two packages are said not to be cloned with each other. One case was for shared package clone detection, and the other was for embedded package clone detection. To establish true negatives for shared package clone detection, 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.

Table 5. Adhoc Detection of 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

To generate true negatives for the embedded package clone detection, we paired up all packages that were reported as being embedded in X, ignoring those cases where

X was the embedded code. This is what we expect our system to report – that X is embedded in Y and Z, but Y is not embedded in Z, and Z is not embedded in Y. In total, we were able to label 9149 negative cases.

5.3 Accuracy of Shared Package Clone Detection

We employed 10-fold validation from our labeled dataset to evaluate the accuracy of our system and experimented with a number of classifiers including Naïve Bayes [11], Multilayer Perceptron, C4.5 [12], and Random Forest [13]. Our results are shown in Table 1 and Table 2. The data is very imbalanced and this skews the accuracy, which easily achieves better than 99%, because we can identify negative cases more easily than positive cases. 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%, the precision is 89.88%, the recall is 70.05%, and the f-measure is 78.73%, 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.

Table 6. Adhoc Detection of 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

Table 7. Automated Vulnerability Inference

TP + FP (Packages)	19
TP (Packages)	10
FP (Packages)	9
TP + FP (CVEs)	132
TP (CVEs)	81
FP (CVEs)	51

Table 8. Automated Detection of Potential Vulnerabilities

Package	Embedded Package
freevo	feedparser
hedgewars	freetype
ia32-libs	* (see text)
libtk-img	tiff
likewise-open	curl
luatex	poppler
planet-venus	feedparser
syslinux	libpng
vnc4	freetype
vtk	tiff

5.4 Accuracy of Embedded Package Clone Detection

We evaluated the embedded package clone detection using a number of classifiers including Naïve Bayes, Multilayer Perceptron, C4.5, and Random Forest. Our results are shown in Table 3 and Table 4. We obtained the best result using the C4.5 classification algorithm. The true positive rate was 75.16%, the false positive rate was 0.75%, the precision was 89.24%, the recall was 75.16%, and the f-measure was 81.60%. We then used this algorithm as a base classifier for our asymmetric bagging meta-classifier with 50 bags. This improved the true positive rate to 91.86% but also increased the false positive rate to 6.72%. We see this as an acceptable trade-off to improve the true positive rate.

5.5 Practical 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.

5.6 Vulnerability Detection

A 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 manually, or using adhoc identification of out-of-date clones. The vulnerabilities in each package we found using clone detection are shown in Table 5 and 6.

5.7 Automated Vulnerability Detection

We performed a more recent evaluation of completely automated vulnerability inference over the years of 2010, 2011, and 2012. Clonewise reported 132 vulnerabilities across 19 packages. We submitted bug reports against each package to Debian Linux. Not all our submitted bug reports were actual vulnerabilities. Some reports were erroneous because Clonewise falsely identified a package clone when one did not exist. Another source of errors was that some bugs we reported as vulnerabilities could not be triggered, even though the clone was correctly identified and had unpatched CVEs. This was true of libpng image processing library being embedded in the syslinux boot loader package. Boot loading displays an image, but does not allow an attacker to control that image to trigger the vulnerability. A high number (64) of vulnerabilities were found in the ia32-libs package. This package contains a list of embedded libraries and is only updated by Debian on point releases. Debian informed us that this package would invariably contain vulnerabilities, but in the unstable release of Debian an alternative approach will be employed which resolves these issues by not embedding libraries.

Debian have not yet confirmed all our bug reports so we investigated each package manually to check that a package clone existed, and that the internal version number of the library was a version vulnerable to the CVE Clonewise reports. The results are shown in Table 7. It should be noted that the high number of true positives is largely accounted for by the 64 vulnerabilities we marked as such once Debian informed us that ia32-libs was by nature collecting vulnerabilities until point releases. Nonetheless, we detected unverified vulnerabilities in more than 50% of the packages Clonewise reported. We performed this manual analysis stage of all vulnerabilities, except for those in ia32-libs, in less than 2 hours. Our results are shown in Table 6. In the case that these potential vulnerabilities are not confirmed by Debian, then Debian will still need to update their internal CVE database to report that those packages are unaffected. Therefore, our work still remains beneficial.

The results of our system demonstrate that we effectively identify vulnerabilities with a false positive rate that is practical for manual verification in a feasible amount of time.

5.8 Clonewise as a Web Service

We have made available some functionality that Clonewise implements at http://www.codeclones.com. The web service takes a tarball of source code and reports if any of around 420 common open source libraries are embedded in it. The web service frontend is implemented in PHP, shell scripts, and Python. The frontend passes the request to HTTP-based load balancer located on another server. The load

balancer then passes the request to a backend cluster. We can scale our system by running a script to add more nodes to the backend cluster as necessary. The web service uses Amazon EC2 to provide the virtual private servers.

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 [14]. 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.

Related works to ours is that of software clone detection [15]. 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. Work has been done on detecting higher-level clones, including file-level clones [4]. Our work extends higher-level clones by being more accurate for package-level duplication. Additionally, clone detection has been used on industrial sources like the Linux kernel [16] or as used by Microsoft engineers [17]. Our system is not as fine grained as traditional code clone detection and detects code similarity at the source file and package level. This allows us to integrate our system into existing practice as can be used 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.

Software plagiarism is another software similarity problem and detection systems for this 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, as in [18]. Structure based techniques rely on using program structure which typically include the use of dependency graphs or parse trees, as in [19] and [20]. Tree and graph edit distances show similarity. [21] and [22] use greedy string tiling. Another approach [23] considers tokenization of source code adaptive sequence alignment.

Clone detection can be performed on the textual stream in a source file once whitespace and comments are removed [24]. 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 [25, 26] 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 [27, 28].

An alternative approach is to use the abstract syntax tree of the source to generate a fingerprint [29]. 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 [30]. An interesting semantic approach to clone detection is to use the memory states of a program [31].

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. In [32], trees were used to represent source code, and subtrees transformed to a vector representation. This allowed for the Euclidean distance and clustering to identify clones. 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. This has advantages in detecting package-level clones without storing all versions of a particular software package.

7 Future Work

Using our classification approach to clone detection, there are several ways we could 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 Sourceforge [33]. 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. 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.

8 Conclusion

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 operating system security and we feel our work is a good step towards this goal.

References

[1] Gailly, J.-L., Adler, M.: zlib (2011), http://zlib.net
[2] Debian Linux (2011), http://www.debian.org
[3] Red_Hat, Fedora Linux (2001), http://fedoraproject.org
[4] Basit, H.A., Jarzabek, S.: A Data Mining Approach for Detecting Higher-Level Clones in Software. IEEE Trans. Softw. Eng. 35, 497–514 (2009)
[5] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explor. Newsl. 11, 10–18 (2009)
[6] Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digital Investigation 3, 91–97 (2006)
[7] Salton, G., Buckley, C.: Term-weighting approaches in automatic text retrieval. Information Processing & Management 24, 513–523 (1988)
[8] Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly (1955)
[9] Japkowicz, N., Stephen, S.: The class imbalance problem: A systematic study. Intell. Data Anal. 6, 429–449 (2002)
[10] Dacheng, T., Xiaoou, T., Xuelong, L., Xindong, W.: Asymmetric bagging and random subspace for support vector machines-based relevance feedback in image retrieval. IEEE Transactions on Pattern Analysis and Machine Intelligence 28, 1088–1099 (2006)
[11] John, G.H., Langley, P.: Estimating continuous distributions in Bayesian classifiers. Presented at the Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence, Montreal, Quebec, Canada (1995)
[12] Quinlan, J.R.: C4.5: programs for machine learning. Morgan Kaufmann Publishers Inc. (1993)
[13] Breiman, L.: Random Forests. Machine Learning 45, 5–32 (2001)
[14] Biedl, C., Adler, M., Weimer, F.: Discovering copies of zlib (2011), http://www.enyo.de/fw/security/zlib-fingerprint/
[15] Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's School of Computing TR 541, 115 (2007)
[16] Jiang, L., Su, Z., Chiu, E.: Context-based detection of clone-related bugs. Presented at the Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia (2007)
[17] Dang, Y., Ge, S., Huang, R., Zhang, D.: Code Clone Detection Experience at Microsoft. In: Proceedings of the 5th International Workshop on Software Clones (2011)
[18] Jones, E.L.: Metrics based plagarism monitoring. Journal of Computing Sciences in Colleges 16, 253–261 (2001)
[19] Son, J.-W., Park, S.-B., Park, S.-Y.: Program Plagiarism Detection Using Parse Tree Kernels. In: Yang, Q., Webb, G. (eds.) PRICAI 2006. LNCS (LNAI), vol. 4099, pp. 1000–1004. Springer, Heidelberg (2006)
[20] Liu, C., Chen, C., Han, J., Yu, P.S.: 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)
[21] Prechelt, L., Malpohl, G., Philippsen, M.: Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science 8, 1016–1038 (2002)
[22] Wise, M.J.: YAP3: improved detection of similarities in computer program and other texts. SIGCSE Bull. 28, 130–134 (1996)

[23] Ji, J.-H., Woo, G., Cho, H.-G.: A source code linearization technique for detecting plagiarized programs. SIGCSE Bull. 39, 73–77 (2007)
[24] Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code, p. 109 (1999)
[25] Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 654–670 (2002)
[26] Livieri, S., Higo, Y., Matushita, M., Inoue, K.: 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 2007), pp. 106–115 (2007)
[27] Li, Z., Lu, S., Myagmar, S., Zhou, Y.: 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 2004), p. 20 (2004)
[28] Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering, 176–192 (2006)
[29] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees, p. 368 (1998)
[30] Krinke, J.: Identifying similar code with program dependence graphs, p. 301 (2001)
[31] Kim, H., Jung, Y., Kim, S., Yi, K.: MeCC: memory comparison-based clone detector. Presented at the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA (2011)
[32] Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. Presented at the Proceedings of the 29th International Conference on Software Engineering (2007)
[33] Geeknet, Sourceforge (2011), `http://sourceforge.net/`