ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions

Jiyong Jang, Maverick Woo, and David Brumley

{jiyongj, pooh, dbrumley}@cmu.edu

1 Unpatched Code Clones

Programmers should never fix the same bug twice. Unfortunately, buggy code often gets copied from project to project and each project fixes the bug independently, which means resources are wasted to diagnose the same bug repeatedly. We call clones of buggy code that has been fixed in only a subset of projects *unpatched code clones*. Unpatched code clones are latent bugs which are likely to be vulnerable and can cause a serious vulnerability window—the time frame between when a vulnerability is disclosed and when a project containing the vulnerable code clone is fixed.

For example, the patch presented in Listing 1 was issued in July 2009 to fix a heap overflow bug in libvorbis. The patched vulnerability can cause a program crash or arbitrary code execution via a maliciously-crafted OGG file [3]. Unfortunately, we found 93 unpatched code clones of this bug in our November 2011 dataset. Projects including mplayer and libtritonus-java in Debian, and mednafen and libvorbisidec in Ubuntu, and ffdshow and guliverkli in SourceForge all had the same vulnerable unpatched code. In this case, a total of 93 packages were exposed to this known vulnerability for over 800 days past the initial patch date.

```
--- a/lib/res0.c
+++ b/lib/res0.c
@@ -208,10 +208,18 @@
    info->partitions=oggpack_read(opb,6)+1;
    info->groupbook=oggpack_read(opb,8);

+   /* check for premature EOP */
+   if(info->groupbook<0)goto errout;
+
    for(j=0;j<info->partitions;j++){
        int cascade=oggpack_read(opb,3);
-       if(oggpack_read(opb,1))
-           cascade|=(oggpack_read(opb,5)<<3);
+       int cflag=oggpack_read(opb,1);
+       if(cflag<0) goto errout;
+       if(cflag){
+           int c=oggpack_read(opb,5);
+           if(c<0) goto errout;
+           cascade|=(c<<3);
+       }
        info->secondstages[j]=cascade;

        acc+=icount(cascade);
```

Listing 1: Patch in libvorbis for CVE-2009-3379

To study how widespread the problem of unpatched code clone truly is and to provide a tool that can help developers fight against it, we developed ReDeBug [5], a system to quickly find unpatched code clones in code bases at the scale of *entire* OS distributions. Using ReDeBug, we examined over 2.1 billion lines of code from all packages in Debian Lenny/Squeeze, Ubuntu Maverick/Oneiric, all C and C++ projects in SourceForge, and also the Linux kernel. ReDeBug identified 15,546 unpatched copies of known vulnerable code from 376 Debian/Ubuntu security-related patches. ReDeBug uses syntax-based pattern matching, which allows it to (i) scale to entire OS distributions, (ii) support many different languages, and (iii) guarantee zero false detections.

- *Scalability:* To give a sense of the scale necessary to find all unpatched code clones, observe that Debian Squeeze alone contains 16 GB of non-empty and non-comment code, spanning over 348 million lines. Using ReDeBug on a machine with a 3.40 GHz i7 CPU and SSD, we were able to scan the 2.1 billion lines of code in our entire dataset against 1,634 buggy code patterns in under 3 hours. With the ability to rapidly search for unpatched code clones, ReDeBug can be used to improve the security of code bases in day-to-day development by promptly checking for copies of known vulnerabilities automatically.

- *Support for many different languages:* OS distributions include programs written in a variety of languages. For example, Debian Squeeze consists of 288 million lines of C/C++, 24 million lines of JAVA, 14 million lines of Python, 12 million lines of Perl, 5 million lines of PHP, and so on. To handle such a large variety of languages, ReDeBug uses a simple, fast, and language-agnostic syntax pattern matching approach to find unpatched code clones. We realize that there are more advanced matching algorithms that are applicable when the code is correctly parsed, and that such algorithms will likely find even more unpatched code clones. The challenge is, however, in the building of robust parsers for each language, which has proven difficult even for professional software assurance companies [1]. While we encourage future developers to add parsing support to ReDeBug, for now ReDeBug opts for simpler robust algorithm that works across a wide variety of languages.
- *Zero false detection rate:* There are two types of false reports any clone detection algorithms can make. The first type is a syntactic "false detection". This happens when an algorithm says an unpatched code clone is present when it is not. ReDeBug eliminates false detections by performing a slower but exact match after all potential matches have been rapidly identified. In contrast, advanced heuristic matching algorithms used to find more code clones can suffer a higher false detection rate. It is important to report only true matches to developers; otherwise, they would end up wasting resources to examine the false reports. The second type is a semantic "false positive". This happens when an algorithm detects an unpatched code clone, but the clone is used in a non-vulnerable way such as when checks have been inserted in earlier locations. Though ReDeBug inevitably can have false positives just like any other syntax-based method, we argue that false positives still present problems because the code can be used in a vulnerable way due to a change in the future.

2 ReDeBug

ReDeBug is available for download as an open source tool on our website `http://security.ece.cmu.edu/redebug/`. ReDeBug is written in Python to make the tool (i) easy to use without the need to compile first, (ii) useful on multiple platforms, and (iii) simple to extend with language-specific optimizations. The website also offers an online unpatched code clone detection service where developers can submit their code to test if it contains known vulnerabilities stored in our database. If a match is found, a report showing both the original buggy code and unpatched code clones found in the submitted code is presented.

```
$ redebug.py -h
usage: redebug.py [-h] [-n NUM] [-c NUM] [-v] patch_path source_path

positional arguments:
  patch_path            path to patch files (in unified diff format)
  source_path           path to source files

optional arguments:
  -h, --help            show this help message and exit
  -n NUM, --ngram NUM   use n-gram of NUM lines (default: 4)
  -c NUM, --context NUM print NUM lines of context (default: 10)
  -v, --verbose         enable verbose mode (default: False)
```

Listing 2: Help message of ReDeBug

A full technical description of ReDeBug has been presented in [5]. Here we concentrate on how to use ReDeBug to find unpatched code clones in practice. As shown in Listing 2, ReDeBug takes two positional arguments: ⟨`patch path`⟩ and ⟨`source path`⟩. The first refers to the top-level patch directory from which we extract original buggy code snippets, and the second points to the top-level directory of the source tree to be checked. As optional arguments, `-n` defines how many lines of code are to be considered as a unit of code to compare, `-c` sets how many surrounding lines of code are to be reported as context, and `-v` enables verbose output.

ReDeBug consists of three major components: (i) `PatchLoader`, which extracts original buggy code snippets from patch files, (ii) `SourceLoader`, which matches source files against known buggy code, and (iii) `Reporter`, which generates a report after performing exact-matching test. We explain each component of ReDeBug with an example of identifying the unpatched code clone for the CVE-2009-3379 vulnerability in the Debian mplayer package.

PatchLoader: ReDeBug takes patch files in the UNIX unified `diff` format, which is popular among open source developers. Listing 1 shows a patch for the CVE-2009-3379 vulnerability in libvorbis in the unified `diff` format. A unified `diff` patch consists of a sequence of `diff` hunks where each hunk includes the filename of a modified file, deleted source code lines that are prefixed by a "-", and inserted source code lines that are prefixed by a "+". Modifications are represented as deletions of old source code lines followed by insertions of new source code lines.

1. Consider a set of patches P_i. ReDeBug extracts original code snippets P_i' from P_i by excluding the lines prefixed by a "+" symbol. This is because the inserted lines are not present in original buggy code. The surrounding context lines

are included to conservatively identify unpatched code clones. ReDeBug requires only the patches but not the pre-patch source code. This allows ReDeBug to save significant space because we do not have to keep the original source code.

2. ReDeBug normalizes the extracted original buggy code P_i' to $\bar{P}_i$ by removing whitespaces except new lines and converting all characters into lower-case. We keep new lines since patches in the unified `diff` format operate at the line level. ReDeBug also identifies file types using the `libmagic` library, and performs language-specific normalization to increase the probability of identifying unpatched code clones. For example, for C, C++, and JAVA, we remove single line comments (`//`), multi-line comments (`/* */`), and curly brackets ({}). The code in Listing 3 shows the normalized buggy code extracted from the code in Listing 1. Regular expressions for such language-specific optimization are defined in `common.py`, which can be easily extended to add more optimizations and support other languages.

```
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);
for(j=0;j<info->partitions;j++)
intcascade=oggpack_read(opb,3);
if(oggpack_read(opb,1))
cascade|=(oggpack_read(opb,5)<<3);
info->secondstages[j]=cascade;
acc+=icount(cascade);
```

Listing 3: Normalized buggy code

3. ReDeBug slides a window of n-lines over the normalized code $\bar{P}_i$. For example, we have 5 windows from the code in Listing 3 when $n = 4$: lines 1–4, 2–5, 3–6, 4–7, and 5–8. For each window w, we apply a list of hash functions H to build a list of hash values $h_i = \{h(w)|w \in \bar{P}_i, h \in H\}$. At present, ReDeBug utilizes 3 hash functions: FNV-1a hash[1], djb2 hash, and sdbm hash[2] (refer to `common.py`). The default context in a `diff` file is 3 lines of code. Therefore, we can guarantee each window has at least 1 changed line by setting $n \geq 4$ (the default n is 4).

SourceLoader: ReDeBug builds a Bloom filter [2] for each source file to check the presence of known vulnerabilities. For example, ReDeBug checks for the CVE-2009-3379 vulnerability in the code in Listing 4 as follows:

```
info->begin=oggpack_read(opb,24);
info->end=oggpack_read(opb,24);
info->grouping=oggpack_read(opb,24)+1;
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);

for(j=0;j<info->partitions;j++){
    int cascade=oggpack_read(opb,3);
    if(oggpack_read(opb,1))
        cascade|=(oggpack_read(opb,5)<<3);
    info->secondstages[j]=cascade;

    acc+=icount(cascade);
}
```

Listing 4: Source code snippet from mplayer package

```
info->begin=oggpack_read(opb,24);
info->end=oggpack_read(opb,24);
info->grouping=oggpack_read(opb,24)+1;
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);
for(j=0;j<info->partitions;j++)
intcascade=oggpack_read(opb,3);
if(oggpack_read(opb,1))
cascade|=(oggpack_read(opb,5)<<3);
info->secondstages[j]=cascade;
acc+=icount(cascade);
```

Listing 5: Normalized source code snippet

1. ReDeBug normalizes source file F_j to $\bar{F}_j$ in a similar way by removing whitespaces except new lines and converting all characters into lower-case. Then, language-specific optimizations such as comment removal are applied according to the identified file type. For example, the code in Listing 4 is normalized into the code in Listing 5.
2. ReDeBug slides a window of n-lines over the normalized source code $\bar{F}_j$. We hash each window w using the same list of hash functions H. Specifically, for each $h \in H$, we set the $h(w)$-th bit of the Bloom filter BF_j to 1. Each source file is now represented by its corresponding Bloom filter.
3. ReDeBug tests if a normalized source file $\bar{F}_j$ includes normalized buggy code $\bar{P}_i$ by checking if every bit in the locations specified by h_i is set to 1 in BF_j. For example, for all the hash values h_i generated from the code in Listing 3, we check if the corresponding bits are set to 1. If at least one of the bits is 0, that means the corresponding window of $\bar{P}_i$ is not present in $\bar{F}_j$. ReDeBug only records the pair $(\bar{P}_i, \bar{F}_j)$ as a potential match if $\bar{F}_j$ contains the entire $\bar{P}_i$.

Reporter: For every pair $(\bar{P}_i, \bar{F}_j)$ recorded, ReDeBug verifies if $\bar{P}_i$ really occurs in $\bar{F}_j$. A Bloom filter may cause false detection due to hash collisions. This is why ReDeBug performs exact match to eliminate any possible false detection due to the use of Bloom filters. For example, the code in Listing 5 indeed contains the buggy code in Listing 3. Finally, ReDeBug reports the Debian mplayer package contains an unpatched code clone of CVE-2009-3379. The report also presents a pair of the patch in Listing 1 and the matched source code in Listing 4, which helps developers to easily inspect the identified unpatched code clone.

[1] http://isthe.com/chongo/tech/comp/fnv/
[2] http://www.cse.yorku.ca/∼oz/hash.html

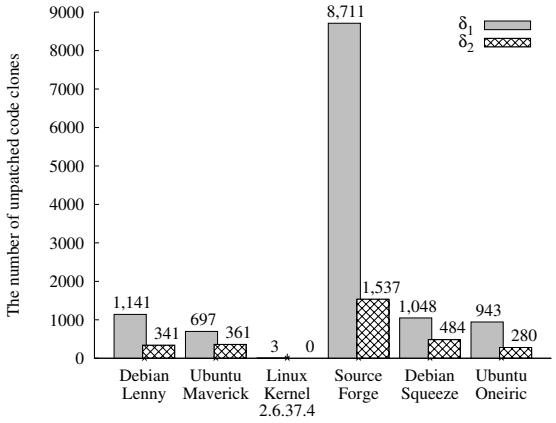

Figure 1: Unpatched code clones in Σ_1 and Σ_2

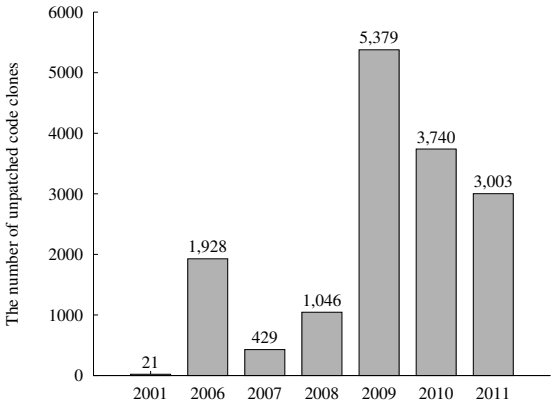

Figure 2: Unpatched code clones from patches in different years

3 Security-Related Bugs

Using ReDeBug, we analyzed over 2.1 billion lines of source code from several OS distributions to comprehend the current trends of unpatched code clones. Table 1 shows the detailed breakup of our collected source code dataset. The Early 2011 dataset (Σ_1) consists of all source packages from Debian 5.0 Lenny, Ubuntu 10.10 Maverick, Linux Kernel 2.6.37.4, and all C/C++ projects in SourceForge. The Late 2011 dataset (Σ_2) contains all source packages from Debian 6.0 Squeeze and Ubuntu 11.10 Oneiric. For the SourceForge packages, we used version control systems such as Subversion, CVS, and Git to obtain up-to-date packages; then we excluded non-active code branches such as `branches` and `tags` directories. For source packages in Debian and Ubuntu, we applied existing patches, e.g., `debian/patches/`, because those patches can be included during a build. As a result, the source packages we checked were patched with all available and included patches on the download date.

Distributions		Lines of Code	Date Collected
Early 2011 (Σ_1)	Debian Lenny	257,796,235	Jan 2011
	Ubuntu Maverick	245,237,215	Mar 2011
	Linux Kernel 2.6.37.4	8,968,871	Mar 2011
	SourceForge (C/C++)	922,424,743	Mar 2011
Late 2011 (Σ_2)	Debian Squeeze	348,754,939	Nov 2011
	Ubuntu Oneiric	397,399,865	Nov 2011
Total		2,180,581,868	-

Table 1: Source code dataset

Dataset	# files	# diffs	Date Released
Pre-2011 Patches (δ_1)	274	1,079	2001–2010
2011 Patches (δ_2)	102	555	2011
Total	376	1,634	-

Table 2: Security-related patch dataset

In order to find security-critical bugs, we collected security-related patches from Debian/Ubuntu security advisories that included the information about the corresponding packages and patches/`diffs`. We downloaded 376 security-related patches whose file names had recognizable CVE numbers, and gathered 1,634 `diffs` from these CVEs. As described in Table 2, pre-2011 patches (δ_1) were available at the time of collecting Σ_1, and 2011 patches (δ_2) were released between the download dates of Σ_1 and Σ_2.

In total, ReDeBug found 15,546 unpatched code clones in the two datasets Σ_1 and Σ_2. Figure 1 shows the detailed breakup of unpatched code clones identified in Σ_1 and Σ_2 when querying for δ_1 and δ_2. We considered three scenarios to understand the current situation of unpatched code clones.

- $\{\delta_1$ & $\delta_2\} \rightarrow \Sigma_1$: The unpatched code clones found in Σ_1 using δ_1 and δ_2 approximate how many (potentially) vulnerable packages an adversary may be able to spot when a patch becomes available. 10,248 unpatched code clones were detected in the SourceForge dataset. The old stable, but still supported on the download date, Debian Lenny and Ubuntu Maverick also had 1,482 and 1,058 unpatched code clones respectively. When security-related bugs are fixed in the original packages, it is important to detect such serious vulnerabilities early before an adversary identifies them.
- $\{\delta_1$ & $\delta_2\} \rightarrow \Sigma_2$: The unpatched code clones identified in Σ_2 using δ_1 and δ_2 roughly indicate how new versions of an OS respond to previously known security vulnerabilities. Debian Squeeze and Ubuntu Oneiric included 1,532 and 1,223 such unpatched code clones respectively. We reported the 1,532 unpatched code clones identified in Debian Squeeze

packages to the Debian security team and package developers. So far, 145 *real bugs* have been confirmed by developers either by private emails or by issuing a patch. This showcases the real world impact of ReDeBug. For some examples of the identified unpatched code clones, please refer to our paper [5] and our website http://security.ece.cmu.edu/redebug/.

- $\delta_1 \rightarrow \Sigma_1$ vs. $\delta_1 \rightarrow \Sigma_2$: We investigated how many unpatched code clones persisted from the previous version of an OS to the latest version of an OS. In our evaluation, we compared the 1,838 unpatched code clones from δ_1 in Σ_1 and the 1,991 unpatched code clones from also δ_1 in Σ_2. Among these 3,829 clones, 1,379 persisted. Figure 2 shows the number of unpatched code clones identified from patches released in different years. Note that 21 of the unpatched code clones are security vulnerabilities that were patched over a decade ago (in 2001). This indicates that unpatched code clones are long-lived in modern OS distributions.

In some cases, unpatched code clones may be found in dead code, e.g., vulnerable code that is present but not included at build time or vulnerable code that is included but never gets executed due to logical conditions. The former usually happens when external library code is embedded in a source package, but the package is written to prefer the available system library to the embedded library. Dead code, however, may still be a latent vulnerability in that the accompanied vulnerable library code can be used depending on the availability of the system library during compilation on the user's machine. For C, specifically, we compile code with an assert statement inserted into the identified buggy code region and look for its corresponding assembly in the binary file to weed out such cases.

```
        else
            *d++ = *src;
-       ++src;
-       --len;
+       if (len > 0) {
+           ++src, --len;
+       }
    }
    *d = '\0';
    return dest;
```

(a) Patch for CVE-2009-4016

```
-   while (*src && (len > 0)) {
+   while (*src && (len > 1)) {
        if(*src & 0x80) {
            *d++ = '.';
            --len;
+           if(len <= 1)
+               break;
        ...
        else
            *d++ = *src;
        ++src;
        --len;
    }
    *d = '\0';
    return dest;
```

(b) Another patch for CVE-2009-4016

Figure 3: Different fix for CVE-2009-4016

ReDeBug may have false positives when unpatched code clone is present but not vulnerable. For example, from the patch for CVE-2009-4016 shown in Figure 3a, an unpatched code clone was detected in `ircd-ratbox` package. The package maintainer informed us that the integer underflow vulnerability was fixed in a different location as shown in Figure 3b which shows two new checks to guard against the vulnerable code. As a result, this unpatched code clone is used in a way that makes it unexploitable. ReDeBug and all other syntax-based approaches share the same problem.

Code Duplication: In order to understand the current situation of code clones, we performed a large scale experiment to measure the overall amount of copied code in OS distributions. We measured this at two different granularities: the function level and the token (n-lines of source code) level.

First, for all C/C++ source files in the Debian Lenny code base, we roughly identified functions using the following Perl regular expression:

```
/^  \w+?\s[^;]*? \( [^;]*?\)\s*({ (?:[^{}]++|(?1))*})/xgsm
```

We realize that an regex may not be able to recognize all functions—that would require a complete parser. However, for our evaluation this is sufficient to provide an estimate of code duplication at the function level. We identified a total of 3,230,554 functions and measured their pairwise similarity using the Jaccard index. As shown in Figure 4, most of the function pairs had very low similarity (below 0.1), which is natural because different packages would have dissimilar code for different functionality. However, surprisingly, 694,883,223 pairs of functions had more than 0.5 similarity, and 172,360,750 of them were more than 90% similar. The result clearly shows a significant amount of code cloning and this suggests that unpatched code clones will continue to be important and relevant in the future.

Second, we calculated the total fraction of shared tokens in each file for the SourceForge dataset. As shown in Figure 5, about 30% of files were almost unique (0–10% shared tokens). In contrast, more than 50% of files shared more than 90% of tokens with other files, which shows that code cloning is common within the SourceForge community as well. Note that 100% of shared tokens in a file does not necessarily mean it is copied from another file as a whole. For example, this could also happen when a file consists of small fractions from multiple files.

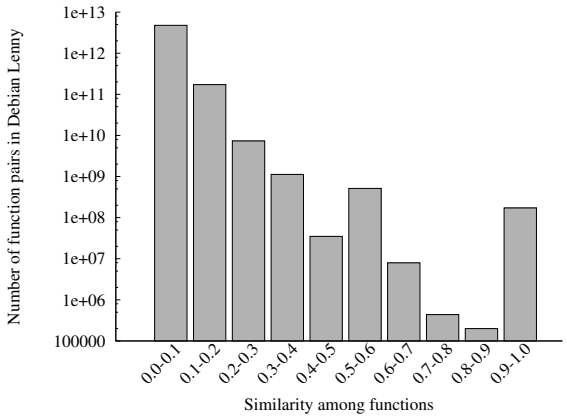

Figure 4: Similarity among functions

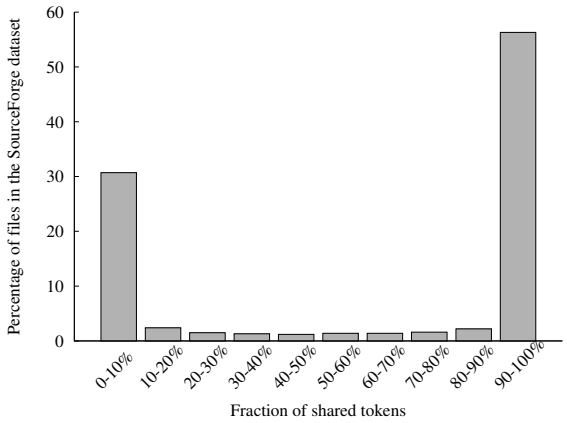

Figure 5: Fraction of shared tokens

4 Related Work

Existing research has focused on finding all code clones, which is a harder problem than just identifying unpatched code clones. Finding all code clones potentially requires comparison among all code pairs, whereas identifying unpatched code clones can be done with a single sweep over the dataset. This line of research uses a variety of matching heuristics based upon high-level code representations such as CFGs and parse trees. For example, CCFinder [7] generates a token sequence from a program using a lexer and transforms the token sequence based on language-dependent rules. A suffix-tree based matching algorithm is then used to determine similar code. CP-Miner [8] parses a program, hashes its tokens into numeric values, and then runs the frequent subsequence mining algorithm to detect clone-related bugs. Deckard [6] and DejaVu [4] both build parse trees and represent structural information of a parse tree as a vector, and then cluster the vectors with respect to the Euclidean distance. An advanced heuristic matching, however, can suffer a higher false detection rate. For example, 73% of bug reports from CP-Miner and 37% of bug reports from DejaVu were false code clones. Furthermore, implementing good parsers is a difficult problem with which even professional software assurance companies struggle [1]. Of course, once that has been done, it will yield a robust level of abstraction that is not available to ReDeBug today.

5 Conclusion

We presented ReDeBug—a system to efficiently detect unpatched code clones. ReDeBug is designed to handle a large code base, e.g., an entire OS distribution written in a wealth of languages. We analyzed over 2.1 billion lines of real code and identified 15,546 unpatched copies of known vulnerable code. This shows that the problem of unpatched code clone is persistent and recurring. The practical impact of ReDeBug has been confirmed by the 145 real bugs that were found and fixed in Debian Squeeze packages. It is our hope that ReDeBug can help developers to enhance the security of their code in day-to-day development.

Acknowledgment

This research was supported in part by sub-award PO4100074797 from Lockheed Martin Corporation originating from DARPA Contract FA9750-10-C-0170 for BAA 10-36. This research was also supported in part by the National Science Foundation through TeraGrid resources provided by Pittsburgh Supercomputing Center. We would like to thank the anonymous referees of our related paper [5] and Debian developers for their feedback in this work.

References

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

[2] Burton H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] National Vulnerability Database. CVE-2009-3379. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3379. Page checked 9/11/2012.

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

[5] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[6] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the international conference on Software Engineering*, 2007.

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

[8] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.