ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions

Jiyong Jang, Abeer Agrawal, and David Brumley
Carnegie Mellon University
Pittsburgh, PA, USA
{jiyongj, abeer, dbrumley}@cmu.edu

***Abstract*—Programmers should never fix the same bug twice. Unfortunately this often happens when patches to buggy code are not propagated to all code clones. Unpatched code clones represent latent bugs, and for security-critical problems, latent vulnerabilities, thus are important to detect quickly.**

In this paper we present ReDeBug, a system for quickly finding unpatched code clones in OS-distribution scale code bases. While there has been previous work on code clone detection, ReDeBug represents a unique design point that uses a quick, syntax-based approach that scales to OS distribution-sized code bases that include code written in many different languages. Compared to previous approaches, ReDeBug may find fewer code clones, but gains scale, speed, reduces the false detection rate, and is language agnostic. We evaluated ReDeBug by checking all code from all packages in the Debian Lenny/Squeeze, Ubuntu Maverick/Oneiric, all SourceForge C and C++ projects, and the Linux kernel for unpatched code clones. ReDeBug processed over 2.1 billion lines of code at 700,000 LoC/min to build a source code database, then found 15,546 unpatched copies of known vulnerable code in currently deployed code by checking 376 Debian/Ubuntu security-related patches in 8 minutes on a commodity desktop machine. We show the real world impact of ReDeBug by confirming *145 real bugs* in the latest version of Debian Squeeze packages.

***Keywords*-debug, unpatched code clone, scalability**

I. Introduction

Patches to buggy code are often not propagated to code clones in real OS distributions. For example, the following patch was issued for Expat, a widely used XML parser to fix a bounds checking bug in August 2009.

```
                                const char *end,
                                POSITION *pos)
    {
-       while (ptr != end) {
+       while (ptr < end) {
            switch (BYTE_TYPE(enc, ptr)) {
    #define LEAD_CASE(n) \
        case BT_LEAD ## n: \
```

Listing 1: CVE-2009-3720

This bug, when exploited, causes a denial of service to the victim [14]. While the above patch fixed Expat in 2009, an additional 386 locations across various Debian, Ubuntu, and SourceForge packages currently have clones of the exact same buggy code, all of which are also likely to be vulnerable. We call such bugs *unpatched code clones*.

In this paper we present ReDeBug, a lightweight syntax-based code clone detection system that identifies unpatched code clones at scale. We have used ReDeBug to analyze entire OS distributions to understand the current situation of unpatched code clones: 1) how much (potentially) vulnerable code can an attacker identify when a patch is released, 2) how responsive is the new version of an OS to known security vulnerabilities, and 3) how many persisting unpatched code clones are from the previous version of an OS to the latest version of an OS.

Existing research has focused on methods for improving the number of code clones detected, e.g., [21, 23–25]. While advancements in finding more code clones is important, current algorithms make several trade-offs:

- *Scalability:* To give a sense of the scale necessary to find all unpatched code clones, consider that Debian Lenny currently contains over 210 million lines of non-empty, non-comment lines of C code alone. Current approaches focus on finding as many code clones as possible, not scale. For example, Deckard [23] was applied to the Linux kernel and JDK, but could not scale to the entire Debian Lenny code base, and started consuming more than 20GB of memory in less than 2 minutes. In big development houses one can use clusters of computers to make the approaches scale [21]. However, solutions that can find a competitive number of code clones without requiring clusters are relevant because they are cheaper to run (thus can be run more often) and are applicable to the wider number of developers who don't readily have distributed computing resources.
- *Lack of support for many different languages:* Since OS distributions include programs written in a variety of languages such as C/C++, Java, Shell, Python, Perl, Ruby, and PHP, we want techniques that are language agnostic. Current research such as Deckard [23], CP-Miner [25], CCFinder [24], and Deja Vu [21] first parse the program and use a variety of matching heuristics based upon high-level code representations such as CFGs and parse trees. However, implementing robust parsers for many different languages is a very difficult problem [7].

- *High false detection rate:* The advanced heuristics used to find more code clones introduce a high false detection rate, i.e., a large number of false code clones are reported. For example, Deja Vu boasts the highest accuracy we are aware of at 26-34%, meaning 66-74% are false code clones. That's 2 out of every 3 reports. A considerable amount of resources would be wasted to inspect all the reported cases.

ReDeBug tackles a new point in the design space where we trade more expensive, yet thorough, pattern matching algorithms for speed, scalability, and a language-agnostic property:

- *Scalability:* ReDeBug uses a syntax-based pattern matching approach that can be implemented using extremely efficient data structures which allow fast querying for code clones when given a patch. ReDeBug processed source code into a database at about 700,000 LoC/min on a commodity desktop. This database of over 2.1 billion lines of code can then be queried in less than 8 minutes. Therefore, ReDeBug can also be used as part of the normal development and patch process on hardware available to an average developer or user, e.g., an average desktop.
- *Language agnostic:* Roughly speaking, ReDeBug performs simple normalization where all whitespace is removed and all characters are transformed to their lower-case equivalent. Such simple normalization allows ReDeBug to identify a variety of latent security vulnerabilities in programs written in many different languages. Interestingly, in our evaluation, Deckard – a tree-based code clone detection technique – *missed* 6x more code clones than ReDeBug, not counting languages handled by ReDeBug but not supported by Deckard (§ IV-A).
- *Lower false detection rate:* ReDeBug focuses on decreasing false detection rate by using a close-to-exact matching instead of fuzzier matching employed by previous code clone work. This means we may find fewer unpatched code clones, but that we will also have fewer false positives due to mis-matches. ReDeBug has false positives when vulnerable code is not detected as dead code by the underlying compiler, and when a previously identified vulnerable code segment is used in a way that makes it non-exploitable. Deja Vu and similar approaches have similar sources of false detection, *plus* errors in the matching algorithms themselves [21, 23–25]. As a result, Deja Vu had a false detection rate of 66-74%, which is on the better end of similar code clone detection mechanisms [21, 23–25]. ReDeBug has zero errors due to matching, thus does not suffer from the major source of false positives found in previous work.

We have used ReDeBug to check for unpatched code clones in Debian 6.0 Squeeze (348,754,939 LoC [1]), Debian 5.0 Lenny (257,796,235 LoC), Ubuntu 11.10 Oneiric (397,399,865 LoC), Ubuntu 10.10 Maverick (245,237,215 LoC), Linux Kernel (8,968,871 LoC), and all C/C++ projects at SourceForge (922,424,743 LoC). So far, ReDeBug has found 15,546 unpatched code clones in the total 2,180,581,868 LoC by checking 376 Debian/Ubuntu security-related patches. The patches address a variety of issues ranging from buffer overflows, to information disclosure vulnerabilities, to denial of service vulnerabilities. Our measurements indicate that even though ReDeBug is simpler, it actually finds a comparable number of code clones to existing approaches (§ IV-A).

Previous work has shown that once a patch is released, an attacker can use the patch to reverse engineer the bug and automatically create an exploit in only a few minutes [10]. Our experiments indicate one security implication of ReDeBug is an attacker, using a single laptop, could potentially find thousands of vulnerable applications among billions of lines of code in only a few minutes once a patch is released, assuming he has already preprocessed the code.

In addition to finding unpatched code clones, we have conducted the first study of the amount of code cloning in the entire Debian Lenny source base. By performing pairwise comparison among functions, ReDeBug provides the distribution of function pairs based upon their similarity (see § III-H)

Overall, our main contributions are:

- We analyze entire OS distributions to comprehend the current trends of unpatched code clones. To the best of our knowledge, ReDeBug is the first tool to explore over 2.1 billion lines of entire OS distributions to understand unpatched code clone problems. We show that unpatched code clones are a recurring problem in modern distributions, and find 15,546 unpatched code clones from Debian Lenny/Squeeze, Ubuntu Maverick/Oneiric distributions, the Linux kernel, and SourceForge. So far, ReDeBug has confirmed 145 real bugs.
- We describe ReDeBug, which suggests a new design space for code clone detection in terms of scalability, speed, and false detection rate. In particular, the design point makes ReDeBug realistic for use by typical developers in everyday environments in order to improve the security of their code by quickly querying known vulnerabilities.
- We provide the first empirical measurement of the total amount of copied code in OS distributions. This suggests that in the future, unpatched code clones will continue to be important and relevant.

[1] We always count non-empty, non-comment lines.

II. REDEBUG

A. The Core System

Finding all unpatched code clones is tricky and involves numerous considerations. For example, how many lines of code need to be similar for a case to be reported? Is one copied line enough, or are we only interested in multiple line matches? Should whitespace matter? Should the order of statements matter, and if so, should we only consider some syntactic classes? Do we consider the syntactic text, tokens, or the parse of files? For example, in C the order of declarations likely does not matter, but the order of computation may. What if two segments are equivalent up to variable naming? What about semantic equivalence, e.g., one code sequence multiples by 2 while the other performs a logical left shift. Are these similar or different?

These questions all involve trade-offs between accuracy, efficiency, and how easy it is to implement a robust algorithm. For example, consider code that is the same up to variable names and variable declaration order. A straight string match of the files may find virtually no commonality. We could certainly address these problems by normalizing declaration order, and parse code to determine variable name equivalence (so-called α-equivalence) [11]. However, running such algorithms require we implement parsing engines (which can be fragile) and run additional algorithms that cost time, thus reducing overall throughput. If we are not careful we may end up subtly analyzing a model of the original program that is not right, e.g., declaration order matters when looking for buggy code clones of incorrect shadow variable declarations.

ReDeBug's choices are motivated by the design space goals of: (1) focusing on unpatched code clones, (2) scaling to large and diverse code bases such as OS distributions, (3) minimizing false detection, (4) being modular when possible and offer a user choice of parameters, and (5) be language-agnostic as much as possible so that we work with the wealth of languages found within an OS distribution code base. The core of the ReDeBug system accomplishes these goals using the following steps:

1) ReDeBug normalizes each file. By default ReDeBug removes typical language comments, removes all non-ASCII characters, removes redundant whitespace except new lines, and converts all characters to lower case. We also ignore curly braces if the file is C, C++, Java, or Perl (as identified by extension or the UNIX `file` command).
   Normalization is modularized so that the exact normalization steps can easily be changed.
2) The normalized file is tokenized based upon new lines and regex substrings.
3) ReDeBug slides a window of length n over the token stream. Each n tokens are considered a unit of code to compare.
4) Given two sets f_a and f_b of n-tokens, we compute the amount of code in common. When finding unpatched code clones, if f_a is the original buggy code snippet we calculate

$$\text{CONTAINTMENT}(f_a, f_b) = \frac{|f_a \cap f_b|}{|f_a|} \quad (1)$$

   When we want to measure the total amount of similarity between files, we calculate the percentage of tokens in common (i.e., the Jaccard index):

$$\text{SIMILARITY}(f_a, f_b) = \frac{|f_a \cap f_b|}{|f_a \cup f_b|} \quad (2)$$

   With either calculation it is common to only consider cases where the similarity or containment is greater than or equal to some pre-determined threshold θ. In our implementation, we also perform obvious optimizations such as when $\theta = 1$ only verifying $f_a \subseteq f_b$ instead of calculating an actual ratio for CONTAINMENT.
5) ReDeBug performs an exact match test on the identified unpatched code clones to remove Bloom filter errors. ReDeBug also uses the compiler to identify when a code clone is dead code when possible.

For example, suppose we have two files $A = t_1t_2t_3t_4$ and $B = t_1t_3t_4t_2$ where each t_i is a token (note tokens are written in the order that they appear in the file). The tokenization is then $A = \{t_1, t_2, t_3, t_4\}$ and $B = \{t_1, t_3, t_4, t_2\}$. When $n = 2$, there are 3 2-token strings in each set: $f_A = \{(t_1, t_2), (t_2, t_3), (t_3, t_4)\}$ and $f_B = \{(t_1, t_3), (t_3, t_4), (t_4, t_2)\}$. The similarity is $1/5$ since 1 out of 5 2-token sets are shared, (t_3, t_4), even though the shared token sequence appears at different places in the file. As a result, ReDeBug works with reordering, insertions, and deletions of up to n-tokens.

ReDeBug is parametrized in two ways: the number of consecutive tokens to consider together, n, and the threshold, θ. n determines the sensitivity for statement reordering, e.g., if $n = 1$ then statement order does not matter at all, $n = 2$ looks at statement pairs, and so on. θ acts as a knob to tell us what is a significant amount of copying. When $\theta = 1$, two files must have exactly the same n-tokens (after normalization). When $\theta = 0$, any match is considered significant. Values in between represent thresholds for the amount of similarity of interest. There is no "right" value for these parameters. In our experiments we show typical values that produce meaningful results. For example, $n = 4$ works well with existing patches.

Design point comparison: Our approach is in stark contrast with current research trends in code clone detection, such as Deckard [23], CP-Miner [25], Deja Vu [21], and others [1, 24], that focus on minimizing missed code clones at the expense of other factors. These approaches also normalize the code, but then perform additional steps such as parsing

```
// Original buggy code
char buf[8];
strcpy(buf, input);
```

```
// Possible patch 1
char buf[8];
- strcpy(buf, input);
+ strncpy(buf, input, 8);
```

```
// Possible patch 2
char buf[8];
+ if(strlen(input) < 8)
  strcpy(buf, input);
```

Figure 1: Buggy code example and two possible patches

the code into high-level representations like parse trees and control flow graphs. They then employ advanced fuzzy matching algorithms on the abstractions to find additional code clones that we may miss. On the other hand, they may report more false code clones, which require significant human effort to inspect all the reported cases. Furthermore, it is known to be very hard to implement good parsers [7].

Overall, the main difference is by employing simpler techniques that is language agnostic, we can focus on efficient data structures and algorithms and ultimately scale to much larger code bases written in many different languages. Our techniques may miss some clones, but minimize false clone detection rate. This is important for at least two reasons. First, by checking all code in a distribution quickly we can make basic guarantees that at least syntactically similar unpatched code clones do not exist. Second, we can conservatively estimate the amount of code cloning in existing large code bases. The more advanced algorithms in the above work have not demonstrated they can make either claim.

B. Unpatched Code Clone Detection

At a high level, there are two approaches to find unpatched code clones in OS distributions: (1) first find all code clones among the source and then check if a patch applies to copies, or (2) check for clones of only the patched code. Previous work has focused on techniques for finding all clones such as in (1). This makes sense when doing bug finding on whole code bases is cheaper to do on unique code snippets. ReDeBug takes approach (2) because we only want to find clones of the original unpatched buggy code.

ReDeBug looks for unpatched code clones where patches are in UNIX unified `diff` format. Unified `diffs` are popular among open source kernel developers, OS distribution maintainers, and are well-integrated into popular revision control systems like Subversion [12].

A unified `diff` patch consists of a sequence of diff hunks. Each hunk contains the changed filename, and a sequence of additions and deletions. Added source code lines are prefixed by a "+" symbol, and deletions are prefixed by a "−" symbol. Line changes are represented as deleting the original line and adding back the changed lines.

The original buggy code includes all code deleted by the patch. However, simply looking for the lines that were changed (by being deleted) is insufficient: we must also consider the surrounding context of the patch.

Consider the buggy code and two possible patch scenarios shown in Figure 1. Patch 1 signifies that `strcpy` is buggy by deleting the line of code. The code is replaced with the safe `strncpy` version. We can go looking for the deleted line of code, and flag it as buggy everywhere we see it. However, patch 2 simply adds a check. Looking for the missing check is not straightforward since we cannot directly look for missing lines of code. Our approach is to look for copies of the surrounding context tokens, c, for each changed line and report clones of the context.

The overall steps used by ReDeBug to detect buggy code clones, shown in Figure 2, are:

- **Step 1: Pre-process the source.** A user obtains all source files used in their distribution. For Debian, this is done using the `apt` tool. ReDeBug then automatically:
  1) Performs normalization and tokenization as described in the core system in § II-A.
  2) Moves an n-length window over the token stream. For each window, the resulting n-tokens are hashed into a Bloom filter.
  3) Stores the Bloom filter for each source file in a raw data format. ReDeBug compresses Bloom filters before storing to disk to save space and to reduce the amount of disk access at query time.

  While initially the above steps would be performed over the entire distribution, day-to-day use would only run the steps on modified files, e.g., as part of a revision control check-in. In our experiments and implementation, we also concatenate per-file Bloom filters for a project into a single bitmap before saving to disk. This is purely an optimization; loading the single large Bloom filter is much quicker than loading a bunch of small Bloom filters on our machines.
- **Step 2: Check for unpatched code copies.** A user obtains a unified `diff` software patch. ReDeBug then automatically:
  1) Extracts the original code snippet and the c tokens of context information from the pre-patch source. The mechanics for the code snippet are simple: we extract lines prefixed by a "-" symbol in the patch (lines prefixed with a "+" symbol are added and thus not present in the original buggy code). If context information is given in the patch, we use

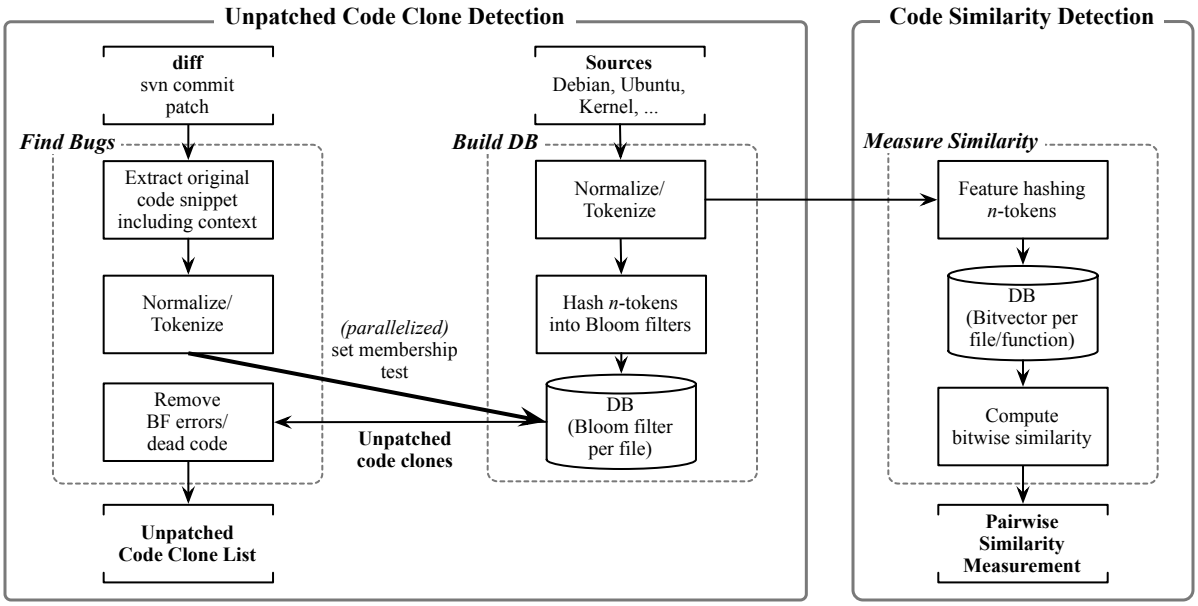

Figure 2: The ReDeBug workflow.

that, else we obtain it from the original source files.

2) Normalizes and tokenizes the extracted original buggy code snippets. The normalization process is the same as described in Step 1. For C, C++, Java, and Python we remove any partial comments in the c context lines since those languages support multi-line comments and c context lines may have only the head or tail part of multi-line comments.
3) Hashes the n-token window into a set of hashes f_p.
4) Performs a Bloom filter set membership test on each hashed n-token window. We report an unpatched code clone with file f if CONTAINMENT$(f_p, f) \geq \theta$.

- **Step 3: Post-process the reported clones.** Given reported unpatched code clones, ReDeBug automatically:
  1) Performs an exact-matching test to remove Bloom filter errors.
  2) Excludes dead code which is not included at build time. For C we add `assert` statements to the buggy code region, and compile with `-g` option which allows us to check the presence of `assert` statements using `objdump -S`. For non-compiled languages this step is omitted.
  3) ReDeBug reports only *non*-dead code to the user.

We use Bloom filters [8] because they are a space efficient randomized data structure used for set membership tests. Suppose there is a data set S, i.e., in our setting a set of n-tokens. A Bloom filter represents set S as a vector of m bits initially all set to 0. To add an element $x \in S$ to the Bloom filter we first apply k independent hash functions of range $\{1..m\}$. For each hash $h(x) = i$, we set the i'th bit of the bit vector to 1. In a Bloom filter, to test if an element of $y \in S$, we again apply the k hash functions and check if all the bits are 1. If at least one of the hashed bits is 0, then we return $y \notin S$. If all bits are set to 1, then we return $y \in S$.

Bloom filters have one-sided error for set membership tests. A false positive occurs when the set membership test returns $x \in S$ when x is not really in S. False positives occur because of collisions in hash functions. The false positive rate of the Bloom filter depends on the size of the bit array (m), the number of hash functions (k), and the number of elements in S (N). The probability of a false positive can be made negligible by an appropriate choice of parameters [9]. Bloom filters have no false negatives. In our setting, the one-sided error means we may mistakenly say that an n-token is present in the set when it is not. The probability of this happening can be made arbitrarily low with appropriate parameter selection, e.g., it is 0.3% in our implementation.

In our evaluation, we only report an unpatched code clone if a file contains all context and all original n-tokens as described above, i.e., $\theta = 1$. This is a conservative configuration.

C. Code Similarity Detection

ReDeBug can also be used to measure the amount of code clones. In this mode ReDeBug uses the SIMILARITY metric between code pairs. The main issue when performing similarity measurements is the cost of the pairwise comparisons.

A standard approach for comparing all N files requires that we compare file 1 to file 2 through N, file 2 to file 3 through N, and so on with a total of $\frac{N(N-1)}{2}$ comparisons.

In order to make pairwise comparisons cheap, we need to implement the Jaccard SIMILARITY set-wise similarity comparisons efficiently. Most standard libraries implement the Jaccard index directly as a set operation on set data container classes, e.g., as done in SimMetrics [4]. The problem is that set data structures are L1/L2 cache inefficient.

ReDeBug uses bitvectors in order to speed up pair-wise comparisons. The idea is that the bitvector operation well-approximates a true Jaccard based upon the original sets. However, the Bloom filters we described above are designed for set membership, not calculating the Jaccard. Jang et al. noticed that as the number of hash functions used in a Bloom filter grows, so does the error in the approximation [22]. The solution is to use a single hash function when creating the bitvectors for SIMILARITY, not multiple hash functions. A single hash function approach is called feature hashing. Jang et al. [22] show that feature hashing well approximates the true Jaccard in both theory and practice (e.g., within 99.99% of the true value with proper parameter selection); we adopt their technique in ReDeBug. A side benefit is that a single hash is faster than multiple hashes used in typical Bloom filter operations.

ReDeBug encodes all elements in a bitvector using feature hashing. ReDeBug then calculates the distance using SIMILARITY_{bv} instead of Equation 2,

$$\text{SIMILARITY}_{bv}(v_a, v_b) = \frac{S(v_a \wedge v_b)}{S(v_a \vee v_b)}$$

where v_i is the bitvector representation of the feature set for file i and $S(\cdot)$ counts the number of set bits.

The complete system for computing similarity for OS-distributions becomes:

1) Obtain all source code. For Debian and Ubuntu, this is done with `apt`. For SourceForge, we crawled all Subversion, CVS and Git directories.
2) For each file, normalize and tokenize as described in § II-A.
3) For each n-length token sequence i, compute $h(i) = d$ and set the d'th bit in the respective file bitvector m.
4) Compute SIMILARITY_{bv} between each pair of bitvectors.

The result is a pairwise similarity measurement between files. In a development environment we would only return those with a similarity greater than θ.

Similarity at different granularities: The source file level provides a relatively coarse granularity of measurement between code bases. ReDeBug can extract functions from files, and compute similarity based upon functions using the same algorithm. ReDeBug can also calculate the total fraction of unique n-tokens. The total fraction of n-tokens suggests the number of unique code fragments found. In our evaluation, we measure both for our data sets (§ III-H).

D. Similarity vs. Bug Finding

The algorithm for finding bugs is similar to that for similarity detection, with a few exceptions. First, we preprocess the source to build a database. Patches are queried against the database while every pair of files is compared when computing similarity. The advantage of this is that the time to build a database and to query bugs increases only linearly as we have more files and patches. The time for similarity detection quadratically goes up with more files due to N^2 comparisons.

The second difference is that we use Bloom filters for unpatched code clone detection, while we used feature hashing for similarity detection. We originally wanted a system that used a single algorithm. While conceptually more elegant, such a design wasn't optimal in either scenario. For example, if we had based our similarity metric on Bloom filters with multiple hash functions, we would have a larger error rate than with feature hashing due to extra collisions from the extra hash functions. If we had used feature hashing instead of Bloom filters, we would again have lower accuracy when performing set membership tests. While this may seem like a subtle difference, previous theoretical and empirical analysis also back up the difference between feature hashing and Bloom filters [22, 29].

Luckily, the internals of implementing both feature hashing and Bloom filters is almost identical. In one case we use a single hash function and have a distance metric interface, and in the other we use multiple hash functions and export a set membership interface.

III. Implementation & Evaluation

A. Implementation

ReDeBug is implemented in about 1000 lines of C code and 250 lines of Python. Normalization is modularized within the Python code. We use the QuickLZ library [3] to perform compression/decompression while setting `QLZ_COMPRESSION_LEVEL` to 3 for faster decompression speed.

B. Unpatched Code Clone Detection Experimental Setup

System Environment: We performed all experiments to find unpatched code clones (both building and querying the database) on a desktop machine running Linux 2.6.38 (3.4 GHz Intel Core i7 CPU, 8GB memory, 256 GB SSD drive). We utilized 8 threads to build a DB and to query bugs.

Dataset: We collected our source code dataset twice: early in 2011 and late in 2011. We first collected our Early 2011 Dataset (Σ_1) in January/March 2011: all source packages for Debian 5.0 Lenny and Ubuntu 10.10 Maverick, as well as all public SourceForge C/C++ projects using version control systems such as Subversion, CVS and Git, and the Linux

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 I: Source Dataset

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

Table II: Security-related Patch Dataset

kernel v2.6.37.4. In the SourceForge data set we removed identifiable non-active code branches such as `branches` and `tags` directories. In November 2011, we prepared our Late 2011 Dataset (Σ_2): all source packages for Debian 6.0 Squeeze and Ubuntu 11.10 Oneiric. Table I shows the detailed breakup of the dataset. The data set consists of a large number of projects written in a wealth of languages including C, C++, Java, Shell, Perl, Python, Ruby, and PHP.

In order to find notable bugs, we collected security-related patches from the Debian/Ubuntu security advisory which has the links to the corresponding packages and patches/`diffs`. We performed our experiments on 376 security-related patches consisting of 1,634 `diffs`. We only included the patches whose related CVE numbers are recognizable by the patch file names. As described in Table II, we downloaded security-related patches released before 2011 (δ_1) which were available at the time of collecting Σ_1, and patches released in 2011 (δ_2) which were distributed between Σ_1 and Σ_2.

In the original source packages for Debian and Ubuntu there are a number of existing patches (e.g., `debian/patches/`) that can be applied during a build; we applied these patches as well. As a result, the packages were patched current up to security advisories on the download date. Since we downloaded the SourceForge packages via revision control systems, we assume all patches were already applied.

The experiments show the number of duplicate buggy code segments that are still likely vulnerable. We have verified the presence of all reported unpatched code clones, i.e., clones of the exact same buggy code, to confirm the ReDeBug implementation is correct. We discuss this measurement in § III-F.

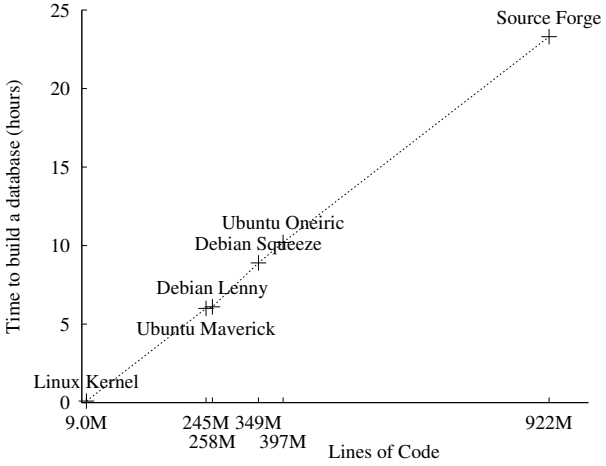

Figure 3: Time to build a database with various sizes of dataset

Distributions	DB Size	Projects #	Files #
Debian Lenny	6.0GB	10,699	1,155,594
Ubuntu Maverick	5.6GB	11,237	1,067,579
Linux Kernel 2.6.37.4	344MB	-	57,653
SourceForge (C/C++)	29GB	30,437	5,574,905
Debian Squeeze	8.2GB	14,977	1,586,325
Ubuntu Oneiric	9.8GB	18,240	1,892,911

Table III: Size of created databases

Default Parameters: The default context in a `diff` file is 3 lines of code. Unless otherwise noted, we set $n = 4$. $n = 4$ when the amount of context $c = 3$ guarantees that every reported duplicate had at least one changed line along with surrounding context. In all experiments for unpatched code clones we set $\theta = 1$, e.g., with the default parameters all n-tokens from the original buggy code segment needed to be found in an unpatched copy to report a bug. m is the size of a Bloom filter and N is the number of n-tokens to be hashed into a Bloom filter. ReDeBug used 256KB-sized Bloom filters where the m/N ratio was always at least greater than 32. ReDeBug took advantage of 3 fast hash functions: djb2, sdbm, and jenkins [6].

C. Performance

We ran ReDeBug to create the database for each source code dataset. Figure 3 shows the database build time. The database build for Ubuntu Maverick and Debian Lenny both took about 6 hours. The database build for SourceForge took about 23.3 hours. This is the end-to-end time including the time to read in files, normalize, tokenize, put into the Bloom filter, and to store on disk for all source code written in a variety of languages, e.g., C/C++, Java, Shell, Perl, Python,

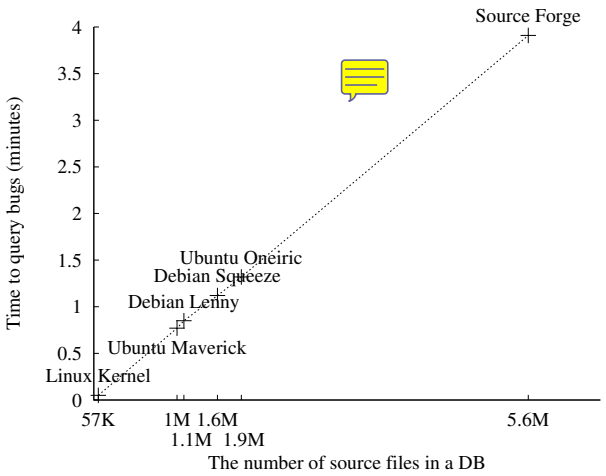

Figure 4: Time to query 1,634 bugs to various sizes of DBs

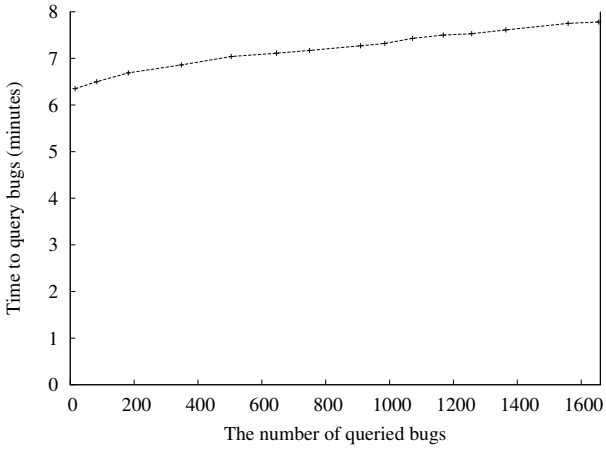

Figure 5: Time to query with various number of bugs against the entire DB

Ruby, and PHP. The experiments suggest that the time to build a database linearly increases as the size of the source code increases. Once ReDeBug has built the initial database, incremental update is quickly done by adding/changing only the relevant parts of the database.

The resulting database sizes and the number of projects and files in databases are described in Table III. As a reference point, Debian Lenny required 282GB to store 1,155,594 files without compression, but only 6.0GB with compression in ReDeBug. The large compression factor is due to the sparseness of the Bloom filters.

Figure 4 depicts the time to query 1,634 security-related patches (δ_1 and δ_2) to each database. As the size of a database (the number of files in a database) grew, the time to query bugs increased linearly. Though there was an overhead to recover compressed Bloom filters to perform the set membership test, the querying time was fast, e.g., 0.04 second per bug against about 1 million source files in the case of Debian Lenny.

Figure 5 shows the time it took to compare a varying number of bugs against the whole database including Σ_1 and Σ_2. The query time has a very gentle upward slope. The results suggest querying even a large number of patches should take only a few minutes. For example, it took about 6 minutes 21 seconds to query 15 `diffs`, and this time increased only slightly to 7 minutes 46 seconds for 1,634 `diffs`.

Together these 3 graphs show that ReDeBug is highly scalable, and it can be applied to find unpatched code copies in day-to-day development. The time it takes us to perform all operations increases linearly with the size of the database, and grows very slowly with the number of `diffs`.

D. Security-Related Bugs

1) $\{\delta_1$ *&* $\delta_2\}$ $\rightarrow$ Σ_1*:* When security-related bugs are fixed in the original projects, all the relevant code clones should also be corrected. In practice, unfortunately, code reuse among open source projects is usually ad-hoc, which makes it difficult to update all relevant projects when the patch is released. An attacker may be able to easily identify the same known vulnerabilities (δ_1 and δ_2) in other projects (Σ_1) that are not patched yet.

We queried δ_1 and δ_2 to Σ_1 to measure how many unpatched code clones are detected, which approximates how many (potentially) vulnerable projects an attacker may be able to spot when a patch becomes available. The total number of unpatched code clones in Σ_1 for δ_1 and δ_2 was 12,791 using the default parameters of $n = 4$ and $c = 3$. The number of matches to each dataset in Σ_1 is shown in Figure 6. The old stable, but still supported, Debian Lenny and Ubuntu Maverick have 1,482 and 1,058 unpatched code clones, respectively.

2) $\{\delta_1$ *&* $\delta_2\}$ $\rightarrow$ Σ_2*:* We measured how many unpatched code clones are identified for δ_1 and δ_2 in Σ_2 consisting of the latest versions of Debian Squeeze and Ubuntu Oneiric. This evaluation demonstrates roughly how responsive the new version of an OS is to previously released security-related patches. Σ_2 still has 1,991 unpatched code clones for δ_1; furthermore, 764 unpatched code clones are reported for δ_2, which indicates that unpatched code clones are recurring in OS distributions.

3) $\delta_1 \rightarrow \Sigma_1$ *vs.* $\delta_1 \rightarrow \Sigma_2$*:* We compared 1,838 unpatched code clones in Σ_1 and 1,991 unpatched code clones in Σ_2 for δ_1; and we found that 1,379 unpatched code clones have persisted. Table IV shows the number of unpatched code clones identified from different years' patches. It is interesting to note that security vulnerabilities that were patched over a decade ago (from 2001) still have 21 unpatched code clones present in Σ_1. Even Σ_2 still have 496 unpatched code clones from 2006 patches. This result demonstrates that unpatched code clones persist in modern distributions.

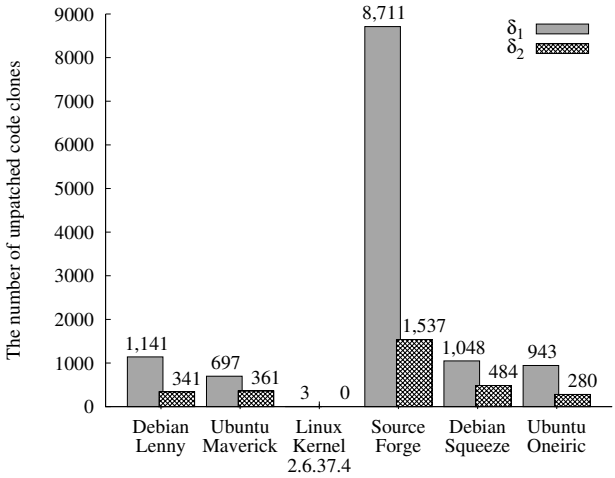

Figure 6: The number of unpatched code clones in Σ_1 and Σ_2

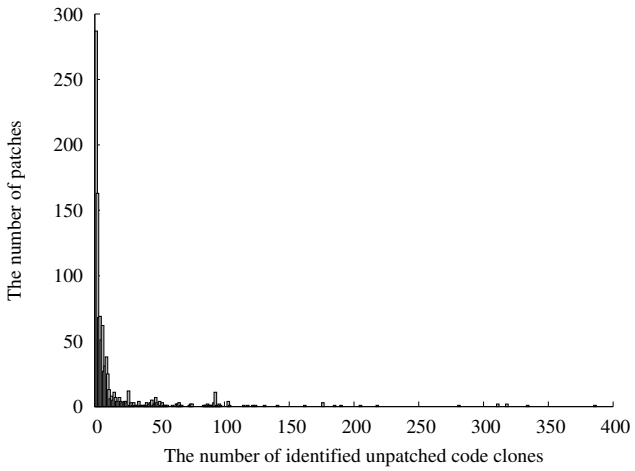

Figure 7: The identified unpatched code clones per patch

	2001	2006	2007	2008	2009	2010	2011
Lenny	2	109	76	88	565	301	341
Maverick	0	161	35	62	248	191	361
Kernel	0	0	0	0	1	2	0
SrcForge	19	1162	227	746	3845	2712	1537
Squeeze	0	264	46	77	379	282	484
Oneiric	0	232	45	73	341	252	280
Total	21	1928	429	1046	5379	3740	3003

Table IV: Unpatched code clones in each distribution from different years' patches

The size of n		$n = 4$	$n = 5$	$n = 7$
# of queried `diffs`		1,634	1,248	503
Unpatched code clones	Debian Lenny	1,482	1,013	309
	Ubuntu Maverick	1,058	736	251
	Linux Kernel 2.6.37.4	3	2	0
	SourceForge (C/C++)	10,248	6,211	2,130
	Debian Squeeze	1,532	1,061	391
	Ubuntu Oneiric	1,223	828	293
Total		15,546	9,851	3,374

Table V: Unpatched code clones with various n for δ_1 and δ_2

E. The Identified Unpatched Code Clones

Figure 7 depicts the distribution of how often we found clones for patches. The maximum was 386 unpatched code clones of the patch shown in Listing 1, with most patches having less than 50 respective unpatched code clones. This result demonstrates that there are potentially many vulnerable code clones for each new patch, motivating the need to implement unpatched code clone detection as part of the developer lifecycle.

Table V shows the number of identified unpatched code clones with various sizes of n. When n increases from 4 to 7, ReDeBug hashes every 7 consecutive tokens and each match represents an exact matching of 7 sequential tokens. Overall, this represents a larger number of tokens exactly matched, which yields a more conservative metric for "real" bugs (see § IV for a discussion).

As we increased n, the total number of unpatched code clones ReDeBug found decreased. Note that as n increases, the total number of `diffs` we queried decreased. The reason is that some `diffs` had fewer than n tokens in total. Overall, in the most conservative experiment, ReDeBug identified 3,374 unique unpatched code copies that likely constitute real bugs.

F. Code Clone Detection Errors

A key question is, what is the false detection rate of ReDeBug? There are several ways to answer this. One popular metric is the accuracy of the matching process. In ReDeBug, this is the Bloom filter tests. The Bloom filter tests have no false negatives, but may have false positives. We performed an exact match test on the 15,599 unpatched code clones initially reported, of which 15,546 were confirmed. Thus, overall we had a 0.3% false positive rate in the Bloom filters. Our post-processing system removes this source of errors from the final output.

In several cases, such as the one shown in Listing 2, the vulnerability was found in dead code, which is not included at build time. This vulnerability can lead to an integer overflow that allows denial of service [15].

```
  es = -1;
  N = 1;
  do {
+       if (N >= 2*1024*1024) RETURN(↩
    BZ_DATA_ERROR);
        if (nextSym == BZ_RUNA) es = es + (0+1) *↩
            N; else
        if (nextSym == BZ_RUNB) es = es + (1+1) *↩
            N;
        N = N * 2;
```

Listing 2: CVE-2010-0405

We matched the above code to libcompress-bzip2-perl. However, the package maintainers stated that the matched code was not an actual vulnerability since it was dead code.

At the post-processing step, we eliminated such dead code which was not included at build time. We measured the number of code clones in *non*-dead code for the 1,354 reported unpatched code clones from 149 Debian Squeeze packages. 831 out of 1,354 (61%) unpatched code clones are confirmed as *non*-dead code, which likely represent real vulnerabilities. Dead code may still be a problem that should be fixed: the accompanied library code may be used depending on users' necessity by flipping compilation flags, and the vulnerability would still likely exist. Nonetheless, we now do post-processing to remove dead code for compiled code from our results.

Note that in the overall system, these errors are ultimately removed, and would never be shown or affect the end user.

G. Examples of Security-Related Bugs

In order to evaluate the practical impact of ReDeBug, we reported 1,532 unpatched code clones identified in Debian Squeeze packages to the Debian security team and developers. So far 145 *real bugs* have been confirmed by developers either in email or by issuing a patch. In this section, we show several examples of the real bugs we found.

Qemu is a processor emulator that can be used as a hosted virtual machine monitor. Various bugs, such as the one in Listing 3, which allows root access on the host machine [13], have been fixed over the past few years. These include CVE-2008-0928 and CVE-2010-2784.

```
   int len, i, shift, ret;
   QCowHeader header;

-  ret = bdrv_file_open(&s->hd, filename, flags)↩
    ;
+  ret = bdrv_file_open(&s->hd, filename, flags ↩
    | BDRV_O_AUTOGROW);
   if (ret < 0)
       return ret;
   if (bdrv_pread(s->hd, 0, &header, sizeof(↩
       header)) != sizeof(header))
```

Listing 3: CVE-2008-0928

These patches were not applied to the derivative package xen-qemu, the Xen version of Qemu. When contacted, Debian and upstream developers confirmed the presence of real bugs and indicated that fixing these bugs was necessary.

The patch in Listing 4 was issued to fix a vulnerability in rsyslog, a Linux and Unix system logger. This vulnerability involved sending a specially crafted log message that leads to denial of service [18].

```
  i = 0;
- while(lenMsg > 0 && *p2parse != ':' && *↩
    p2parse != ' ' && i < CONF_TAG_MAXSIZE) {
+ while(lenMsg > 0 && *p2parse != ':' && *↩
    p2parse != ' ' && i < CONF_TAG_MAXSIZE - 2) ↩
    {
    bufParseTAG[i++] = *p2parse++;
     --lenMsg;
```

Listing 4: CVE-2011-3200

This patch was not applied to the Debian package rsyslog-gssapi, a version of rsyslog with plugins that allowed rsyslog to write and receive GSSAPI encrypted logging messages. The package maintainers when contacted decided to fix the vulnerability by issuing an update.

The patch below was issued to fix a heap based buffer overflow vulnerability in the Paint Shop Pro plugin in GIMP 2.6.11 [17].

```
-       if (code >= max_code)
+       if (code == max_code)
          {
-           *sp++ = firstcode;
+           if (sp < &(stack[STACK_SIZE]))
+             *sp++ = firstcode;
            code = oldcode;
          }

-       while (code >= clear_code)
+       while (code >= clear_code && sp < &(stack↩
    [STACK_SIZE]))
          {
            *sp++ = table[1][code];
            if (code == table[0][code])
```

Listing 5: CVE-2011-1782

When contacted, the developers of Deutex, a Debian package used to manipulate files for various games, indicated that this was likely a real vulnerability.

The following patch was issued to fix an integer overflow in PHP before 5.3.6 which could lead to a denial of service and possibly information leak [16]. This patch was not employed to the Debian PHP package. After contacted, the package maintainer issued a patch to fix the bug.

```
- if (start + count > shmop->size || count < 0) ↩
    {
+ if (count < 0 || start > (INT_MAX - count) || ↩
    start + count > shmop->size) {
    php_error_docref(NULL TSRMLS_CC, E_WARNING, ↩
        "count is out of range");
    RETURN_FALSE;
```

Listing 6: CVE-2011-1092

Listing 7 shows a recent patch for CVE-2011-3145, which was successfully patched in an Ubuntu Oneiric package, but not in a Debian Squeeze package. This patch fixed an incorrect `/etc/mtab` ownership error in the ecryptfs-utils package, which might cause to unmount arbitrary location [27].

```
      if (setreuid(uid, uid) < 0) {
        perror("setreuid");
      }
+     if (setregid(gid, gid) < 0) {
+       perror("setregid");
+     }
      goto fail;
    }
  } else {
```

Listing 7: CVE-2011-3145

After we contacted the developers, the same patch applied to the Ubuntu Oneiric package was issued for the Debian Squeeze package to fix the vulnerability.

Listing 8 shows a security patch applied to the Ubuntu Oneiric apache2 package to fix the vulnerability where remote attackers can send requests to intranet servers with a well-crafted URI [19]. The same patch was applied to the Debian Squeeze package to fix the bug after we reported it.

```
 ap_parse_uri(r, uri);

+/* RFC 2616:
+ *   Request-URI    = "*" | absoluteURI | abs_path | authority
+ *
+ * authority is a special case for CONNECT.  If the request is not
+ * using CONNECT, and the parsed URI does not have scheme, and
+ * it does not begin with '/', and it is not '*', then, fail
+ * and give a 400 response. */
+if (r->method_number != M_CONNECT
+ && !r->parsed_uri.scheme
+ && uri[0] != '/'
+ && !(uri[0] == '*' && uri[1] == '\0')) {
+ ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, r,
+     "invalid request-URI %s", uri);
+ r->args = NULL;
+ r->hostname = NULL;
+ r->status = HTTP_BAD_REQUEST;
+ r->uri = apr_pstrdup(r->pool, uri);
+}
+
 if (ll[0]) {
  r->assbackwards = 0;
  pro = ll;
```

Listing 8: CVE-2011-3368

In § I, we motivated the need to handle many languages. Here, we show a non-C example in Ruby. Listing 9 shows a security patch for the puppet package to fix the vulnerability where an attacker can impersonate a master by exploiting a non-default `certdnsnames` option when generating certificates [20]. After we reported this bug, the package maintainer fixed the vulnerability by issuing a security patch.

```
 # Sign a given certificate request.
-def sign(hostname, cert_type = :server, self_signing_csr = nil)
+def sign(hostname, allow_dns_alt_names = false, self_signing_csr = nil)
   # This is a self-signed certificate
   if self_signing_csr
+    # # This is a self-signed certificate, which is for the CA.  Since this
+    # # forces the certificate to be self-signed, anyone who manages to trick
+    # # the system into going through this path gets a certificate they could
+    # # generate anyway.  There should be no security risk from that.
     csr = self_signing_csr
+    cert_type = :ca
     issuer = csr.content
   else
+    allow_dns_alt_names = true if hostname == Puppet[:certname].downcase
     unless csr = Puppet::SSL::CertificateRequest.find(hostname)
       raise ArgumentError, "Could not find certificate request for #{hostname}"
     end
+
+    cert_type = :server
     issuer = host.certificate.content
+
+    # Make sure that the CSR conforms to our internal signing policies.
+    # This will raise if the CSR doesn't conform, but just in case...
+    check_internal_signing_policies(hostname, csr, allow_dns_alt_names) or
+      raise CertificateSigningError.new(hostname), "CSR had an unknown failure checking internal signing policies, will not sign!"
   end

   cert = Puppet::SSL::Certificate.new(hostname)
-  cert.content = Puppet::SSL::CertificateFactory.new(cert_type, csr, content, issuer, next_serial).result
+  cert.content = Puppet::SSL::CertificateFactory.
+    build(cert_type, csr, issuer, next_serial)
   cert.content.sign(host.key.content, OpenSSL::Digest::SHA1.new)
```

Listing 9: CVE-2011-3872

H. Copied Similarity Metrics

In order to understand the overall amount of code duplication, we also ran a large scale experiment to measure the similarity within the entire Debian Lenny source base. These experiments required the pairwise computations discussed in § II-C. We ran these experiments on an SGI UV 1000 cc-NUMA shared-memory system consisting of 256 blades. Each blade has 2 Intel Xeon X7560 (Nehalem) eight-core processors and 128 Gbytes of local memory [2].

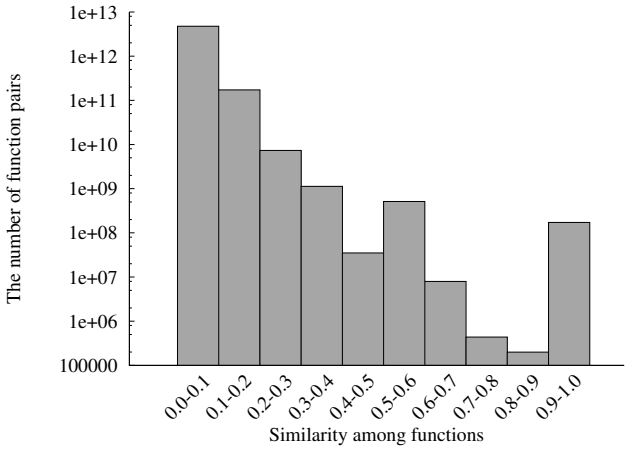

Figure 8: Similarity among functions

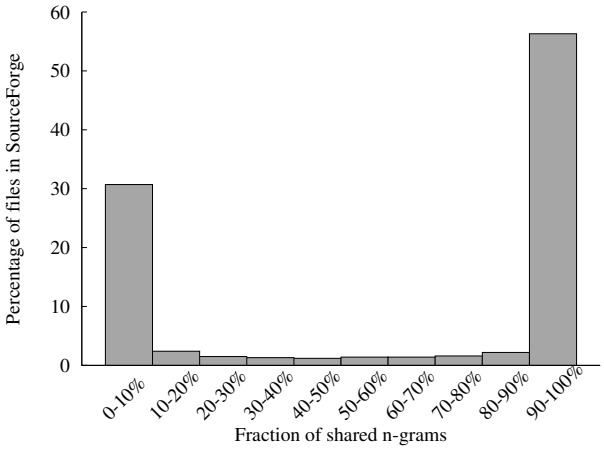

Figure 9: Fraction of shared n-grams

For non-C/C++ code we normalized as described in § II-C. For C/C++ code, we also did experiments where we roughly identified functions within C/C++ files using the following Perl regular expression:

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

The regex isn't perfect to recognize all functions since that would require a complete parser. However, in our experience it is sufficient and allowed us to provide an estimate of similarity and code clones at the function level.

We identified 3,230,554 functions containing at least 4 tokens. We split identified functions into two groups based upon the function size. "The small-sized" group had 3,144,998 functions which had less than 114 tokens. "The large-sized" group had 85,556 functions. Overall, we measured pair-wise distance in each group using SIMILARITY$_{bv}$, which required 4,949,164,509,293 pairwise comparisons.

For the small-sized group, we used 32 byte bit vectors. Total bitvector generation time was 6 min using 32 CPUs. It took 19 min to compare every pair of the group using 512 CPUs. For the large-sized group, we used 8 KB bit vectors. Generation time was 14 minutes on 32 CPUs. Pairwise comparisons took 5 min 30sec using 512 CPUs.

Figure 8 shows the distribution of function pairs based upon their similarity. Most of the function pairs had very low similarity below 0.1, which is natural in that different packages would be expected to have different functionality. However, surprisingly, 694,883,223 pairs of functions had more than 0.5 similarity. Among them, 172,360,750 pairs were more than 90% similar. This result clearly shows a significant amount of code copying.

While performing our experiments, we noticed that the SourceForge dataset had more code clones. With SourceForge projects, we normalized and tokenized each file and calculated the total fraction of shared n-tokens in each file. As shown in Figure 9, more than 50% of files shared more than 90% of n-tokens with other files. Note that 100% of shared n-tokens in a file does not necessarily mean it is copied from another file as a whole. This could also happen when a file consists of small fractions from multiple files. On the contrary, about 30% of files were almost unique (0-10% shared tokens) while 50% of the files shared more than 90% of all tokens. This shows that code cloning is active and alive within the SourceForge community.

IV. Discussion

A. Comparison to Prior Work

ReDeBug improves scalability with decreased false detection rate, but may find fewer code clones than previous code clone detection work. In order to measure the number of unpatched code clones that ReDeBug missed, we compared the number of code clones detected by ReDeBug to the number of code clones reported by Deckard [23]. We chose Deckard because it claims better code clone detection performance than CP-Miner [25] and CloneDR [5].

Theoretically, the code clones reported by Deckard should be the superset of the code clones found by ReDeBug. In practice, however, Deckard missed more code clones than ReDeBug. We used Deckard v1.2 [2] for our experiments, and set parameters as follows: `minT` (minimum number of tokens required for clones) = 30 and `stride` (size of the sliding window) = 2 for their conservative results, and `Similarity` = 1 to minimize their false detection. This was similar to the setup in their paper.

Deckard did not scale to the entire Debian Lenny distribution (257,796,235 LoC) in our test setup. During pair-

[2] https://github.com/skyhover/Deckard

Clone Detection	Real Clones	False Detection	Missed
ReDeBug	180	0	15
Deckard	96	183	99

Table VI: Code clone detection performance

wise comparisons Deckard consumed more than 20 GB of memory in less than 2 minutes, after which we killed the process. Instead of the entire OS, we ran Deckard on each package at a time with 28 randomly selected C code files which contain security bugs. We report only code clones which match with the buggy code regions. Deckard took more than 12 hours to complete the code clone detection in Debian Lenny, utilizing 8 threads to process 8 packages at the same time. While Deckard processed only the source code written in C (Deckard can process one language at a time), ReDeBug processed a wealth of languages (e.g., C/C++, Java, Shell, Python, Perl, Ruby, and PHP) in 6 hours.

Table VI shows the code clone detection results of Deckard and ReDeBug. As expected, ReDeBug had no false detections, and surprisingly, missed 6 times as few code clones compared to Deckard. The code clones that ReDeBug missed came from the use of different variable names or types.

Deckard faired worse than ReDeBug despite using a more sophisticated strategy. We investigated the causes, and found that 38 out of 99 of the cases were due to parse failures in Deckard, with the remainder just being missed due to the algorithm for detecting code clones. This result lends support that parsing code is hard and can be a limiting factor in practice, and that ReDeBug's relatively simpler approach can be valuable in such circumstances.

B. Unpatched code clones that are not vulnerable

Since ReDeBug gets rid of Bloom filter errors and dead code, a metric for false positives is the number of unpatched code clones that were not vulnerable for some other reason. We have identified two other causes for this type of false positive. First, normalization may be too aggressive in some circumstances and thus the identified code clone is not really a code clone. Second, we may find real unpatched code clones, but other code modifications may prevent the unpatched code from being called in an exploitable context.

Normalization reduces the false negative rate, but may increase the false positive rate. For example, imagine two code sequences that are equivalent but one is performed on an unsigned integer "A" and the other on a signed integer "a". If the bug relates to signedness, only the latter code is vulnerable. However, normalization converts all variables to lower-case, thus we would mistakenly report the former as also buggy.

Listing 10 shows an example of an unpatched code clone that is present but not vulnerable. The patch fixes an integer signedness bug in various BSD kernels. NetBSD contains the same vulnerable code, but fixed the problem by changing the type of `crom_buf->len` from signed integer to unsigned integer instead of using the shown patch.

```
- if (crom_buf->len < len)
+ if (crom_buf->len < len && crom_buf->len > 0)
```

Listing 10: CVE-2006-6013

An unpatched code clone was detected in the ircd-ratbox package from the patch shown in Listing 11. The package maintainer informed us that the vulnerability was fixed in a different location, i.e., adding a separate error checking routine `if (len<=1) break;` ahead of the vulnerable code.

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

Listing 11: CVE-2009-4016

V. Related Work

MOSS [28] is a well-known similarity detection tool using n-tokens. MOSS is based upon an algorithm called winnowing [28], a fuzzy hashing technique that selects a subset of n-tokens to find similar code. The main difference is that ReDeBug uses feature hashing to encode n-tokens in a bitvector, which allows ReDeBug to perform similarity comparison in a cache-efficient way. We swap out winnowing for feature hashing for improved speed. This decision was based upon the work by Jang et al. [22]. Furthermore, in order to find unpatched code clones we use the insight of only looking for code clones of patched bugs to scale to large OS distributions.

Most recent work in academia has focused on detecting all code clones (i.e., reducing the number of missed code clones, but having more false detections of clones). Examples include Deckard [23], CCFinder [24], CP-Miner [25] and Deja Vu [21]. Detecting all code clones is a harder problem than just searching for copies of patched code in that the former potentially requires comparison of all code pairs, while the latter is a single sweep over the data set. 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 uses lexing and then performs transformations based upon rules to determine whether code is similar [24]. The transformation rules are

language-dependent. Deckard [23] and Deja Vu [21] build parse trees for C code, and then reduce part of the parse trees to a vector. Comparisons are done on the vector. CP-Miner also parses the program: currently the parser is implemented only for C and C++. It then hashes these tokens and assigns a numeric value to each, and runs the CloSpan frequent subsequence mining algorithm to detect code clones.

Each of the above techniques represent a unique and different point in the design space. For example, building the parse tree, CFG, etc. all require implementing a robust parser for the language. Implementing good parsers is a difficult problem with which even professional software assurance companies struggle [7], but once done will give them a robust level of abstraction not available to ReDeBug. The highest false positive rate for reported errors among the code clones was 90% for CP-Miner, and 66-74% for Deja Vu. In terms of scalability, the largest code base we are aware of is Deja Vu, which looked at a proprietary code base consisting of about 75 million lines of C code. They used a cluster of 5 machines, and integrated the product into the build cycle. It found 2070-2760 likely bugs. Our experiments are on code bases upon to billions of lines of code (two orders of magnitude larger). If their techniques could be scaled up, they would likely find more unpatched code clones, but the number of falsely detected clones would also scale up and the overall system would require more resources than available to a typical end-developer.

SYDIT [26] is a program transformation tool, which characterizes edits as AST node modifications and generates context-aware edit scripts from example edits. It was tested on an oracle data set of 56 pairs of example edits from open source projects in Java. SYDIT complements ReDeBug in that SYDIT looks at abstract, semantic changes while ReDeBug focuses on syntactic changes at large scale.

Pattern Insight's Code Assurance [1] (aka Patch Miner) is advertised as finding unpatched code clones, as with ReDeBug. Their whitepaper does not contain any technical information to compare on a usage, algorithmic, performance, or accuracy basis, but does mention it performs a kind of "fuzzy matching". We have contacted Pattern Insight to get more details, but they have not made the product available to us for comparison.

Previous work has also done clustering. Much previous work, like us, uses the Jaccard distance metric, e.g., Deja Vu. Deckard and Deja Vu use locality sensitive hashing (LSH) to speed up the pairwise comparison using Jaccard. We use feature hashing. Theoretical analysis shows feature hashing outperforms LSH alone [29], and Jang et al. back this up with an empirical evaluation for malware clustering [22]. However, a hybrid approach that first uses LSH to find near-duplicates which are then compared using feature hashing may be possible. We leave these types of optimizations as future work.

Brumley et al. have shown once a patch becomes available, an attacker may be able to use it to reverse engineer the problem and create an exploit automatically [10]. We leave exploring the ramification of this problem as future work.

VI. Conclusion

In this paper we presented ReDeBug, an architecture designed for unpatched code clone detection. ReDeBug was designed for scalability to entire OS distributions, the ability to handle real code, and minimizing false detection. ReDeBug found 15,546 unpatched code clones, which likely represent real vulnerabilities, by analyzing 2.1 billion lines of code on a commodity desktop. We demonstrate the practical impact of ReDeBug by confirming 145 real bugs in the latest version of Debian Squeeze packages. We believe ReDeBug can be a realistic solution for regular 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, Debian developers, Spencer Whitman, Edward Schwartz, JongHyup Lee, Yongsu Park, Tyler Nighswander, and Maverick Woo for their feedback in preparing this paper.

References

[1] Code Assurance. http://patterninsight.com/products/code-assurance/. Page checked 3/4/2012.

[2] Pittsburgh Supercomputing Center. http://www.psc.edu/. Page checked 3/4/2012.

[3] QuickLZ. http://www.quicklz.com/. Page checked 3/4/2012.

[4] SimMetrics. http://sourceforge.net/projects/simmetrics/. Page checked 3/4/2012.

[5] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering*, 2004.

[6] Daniel Bernstein. http://www.cse.yorku.ca/~oz/hash.html. Page checked 3/4/2012.

[7] 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 Association for Computing Machinery*, 53(2):66–75, 2010.

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

[9] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[10] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[11] Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theoretical Computer Science*, 403:285–306, 2008.

[12] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version control with subversion. http://svnbook.red-bean.com/en/1.0/svn-book.html#svn-ch-3-sect-4.3.2. Page checked 3/4/2012.

[13] National Vulnerability Database. CVE-2008-0928. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0928. Page checked 3/4/2012.

[14] National Vulnerability Database. CVE-2009-3720. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3720. Page checked 3/4/2012.

[15] National Vulnerability Database. CVE-2010-0405. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0405. Page checked 3/4/2012.

[16] National Vulnerability Database. CVE-2011-1092. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1092. Page checked 3/4/2012.

[17] National Vulnerability Database. CVE-2011-1782. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1782. Page checked 3/4/2012.

[18] National Vulnerability Database. CVE-2011-3200. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3200. Page checked 3/4/2012.

[19] National Vulnerability Database. CVE-2011-3368. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3368. Page checked 3/4/2012.

[20] National Vulnerability Database. CVE-2011-3872. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3872. Page checked 3/4/2012.

[21] 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.

[22] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[23] 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.

[24] 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.

[25] 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.

[26] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: generating program transformations from an example. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.

[27] Ubuntu Security Notice. CVE-2011-3145. http://www.ubuntu.com/usn/usn-1196-1/. Page checked 3/4/2012.

[28] Saul Schleimer, Daniel Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD/PODS Conference*, 2003.

[29] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and S.V.N. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10:2615–2637, 2009.