Rendezvous: A Search Engine for Binary Code

Wei Ming Khoo
University of Cambridge, UK
Wei-Ming.Khoo@cl.cam.ac.uk

Alan Mycroft
University of Cambridge, UK
Alan.Mycroft@cl.cam.ac.uk

Ross Anderson
University of Cambridge, UK
Ross.Anderson@cl.cam.ac.uk

***Abstract*—The problem of matching between binaries is important for software copyright enforcement as well as for identifying disclosed vulnerabilities in software. We present a search engine prototype called Rendezvous which enables indexing and searching for code in binary form. Rendezvous identifies binary code using a statistical model comprising instruction mnemonics, control flow sub-graphs and data constants which are simple to extract from a disassembly, yet normalising with respect to different compilers and optimisations. Experiments show that Rendezvous achieves F_2 measures of 86.7% and 83.0% on the GNU C library compiled with different compiler optimisations and the GNU coreutils suite compiled with gcc and clang respectively. These two code bases together comprise more than one million lines of code. Rendezvous will bring significant changes to the way patch management and copyright enforcement is currently performed.**

I. Introduction

Code reuse is a common practice in software engineering. A search for the phrase "based on" on the popular social coding site Github [1] revealed 269,789 hits as of February 2013. Mockus [2] found that 50% of all files created by open source projects were reused at least once in another open source project; the most reused files amongst these projects were from the GNU C library and the Linux kernel. Sometimes code is reused without due consideration to its licensing terms. As of 2012, Welte reported 200 cases of GNU public license violations in software products that were either successfully enforced or resolved by gpl-violations.org [3]. Code reuse also comes with the risk that a vulnerability disclosed in the imported code becomes an undisclosed vulnerability in the main product.

Currently, an auditor seeking to identify reused components in a software product, such as for GPL-compliance, has two main approaches available: source code review and software reverse engineering. However, source code availability is not always guaranteed, especially if the code was developed by a third party, and understanding machine code is at present rather challenging.

Our proposed approach is to bootstrap the process by providing a search engine for binary code. By leveraging on open source initiatives such as GNU, the Apache foundation, Linux and BSD distributions, and public code repositories such as Github and Google code [4], we can reframe identifying code reuse as an indexing and search problem, and a good solution to this problem will entail a provenance service for binary code.

As of February 2013, the GNU C library has an estimated 1.18 million lines of code, the GNU core utilities tool suite has an estimated 57,000 lines of C, and the Linux kernel has 15.3 million lines according to the open source project tracker Ohloh [5]. With this many lines of code we have today, it is clear that we need an efficient method for indexing and search. Ideally we would like both speed and accuracy, so a reasonable approach is to initially approximate using several existing methods that are fast, then optimise and evaluate for accuracy. Another viable approach is to have accuracy as a goal, then optimise for speed. We have adopted the former approach here.

Thus, our primary goal is *efficiency*: We want to have a fast information retrieval scheme for executable code. Like a text search engine, having speedy response times are key for usability.

Our secondary goals are *precision* (low false positives) and *recall* (low false negatives). Since we cannot control the compiler or the options used, the model has to be robust with respect to compilation and their various optimisations.

This paper makes the following contributions.

1) We address the problem of identifying large code bases generated by different compilers and their various optimisations which, to the best of our knowledge, has not been done before.
2) We have implemented a prototype search engine for binary code called Rendezvous, which makes use of a combination of instruction mnemonics, control flow sub-graphs and data constants (Section III). Experiments show that Rendezvous is able to achieve a 86.7% F_2 measure (defined in Section VII) for the GNU C library 2.16 compiled with `gcc -O1` and `-O2` optimisation levels, and an 83.0% F_2 measure for the coreutils 6.10 suite of programs, compiled with `gcc -O2` and `clang -O2` (Section X).
3) As an early prototype, the efficiency of Rendezvous is about 0.407 seconds per function in the worst case. (Section X-H).

II. Design Space

Since code presented to our retrieval system may be modified from the original, we would like to be able to identify the invariant parts of it, such as certain unique functions. This can be accomplished by approximating executable code by a statistical model. A statistical model comprises breaking up code into short chunks, or *tokens*, and assigning a probability to their occurrence in the reference corpus.

978-1-4673-2936-1/13/$31.00 © 2013 IEEE

MSR 2013, San Francisco, CA, USA

TABLE I
A SUMMARY OF THE DIFFERENT ABSTRACTIONS CONSIDERED.

	Abstraction
1	Instruction mnemonic n-grams [9]
2	Instruction mnemonic n-perms [10]
3	Control flow sub-graph [11]
4	Extended control flow sub-graph
5	Data constants

What should the tokens consist of? For natural language, the choice of token is a word or phrase; machine code is more complex in comparison. The decision space of program abstractions is not unexplored, and the typical choice to make is between static and dynamic methods. We chose in favour of static analysis primarily for its relative simplicity. Dynamic analysis has the disadvantage that it requires a virtual execution environment, which is costly in terms of time and resources [6]. Static disassembly is by no means a solved problem in the general sense [7], but there are well-known techniques for it, such as linear sweep and recursive traversal [8], and they work well for a wide range of executables.

Three candidate abstractions, or representations, were considered—instruction mnemonics, control flow sub-graphs, and data constants. Instruction mnemonics, refers to the machine language instructions that specify the operation to be performed, for example `mov`, `push` and `pop` instructions on the 32-bit x86 architecture. A control flow graph (CFG) is a directed graph that represents the flow of control in a program. The nodes of the graph represent the basic blocks; the edges represent the flow of control between nodes. A sub-graph is a connected graph comprising a subset of nodes in the CFG. Data constants are fixed values used by the instructions, such as in computation or as a memory offset. The two most common types of constants are integers and strings. These abstractions were chosen as they are derived directly from a disassembly. Instruction mnemonics was chosen as the simplest abstraction for code semantics; the control flow graph was chosen as the simplest abstraction for program structure; data constants were chosen as the simplest abstraction for data values. A summary of the types of abstractions we considered is given in Table I.

Fig. 1. Overview of the code abstraction process.

III. CODE ABSTRACTION

The code abstraction procedure involves three steps (Figure 1). The executable is first disassembled into its constituent *functions*. The disassembled functions are then *tokenised*, that is, broken down into instruction mnemonics, control-flow sub-graphs and constants. Finally, tokens are further processed to form *query terms*, which are then used to construct a search query.

```
01   // a: array, n: size of a
02   void bubblesort( int *a, int n ) {
03      int i, swapped = 1;
04      while( swapped ) {
05         swapped = 0;
06         for( i = 0; i < n-1; i++ ) {
07            if( a[i] < a[i+1] ) {
08               int tmp = a[i];
09               a[i] = a[i+1];
00               a[i+1] = tmp;
11               swapped = 1;
12            }
13         }
14      }
15   }
```

Fig. 2. Source code for our running example: bubblesort.

We will now describe in detail the abstraction process for the three different abstractions using a running example. Figure 2 shows the sorting algorithm *bubblesort* written in C.

IV. INSTRUCTION MNEMONICS

We make use of the Dyninst binary instrumentation tool [12] to extract the instruction mnemonics. An instruction mnemonic differs from an opcode in that the former is a textual description, whilst the latter is the hexadecimal encoding of the instruction and is typically the first byte. Multiple opcodes may map to the same mnemonic, for instance, opcodes `0x8b` and `0x89` have the same mnemonic `mov`. Dyninst recognises 470 mnemonics, including 64 floating point instructions and 42 SSE SIMD instructions. We used 8 bits to encode the mnemonic, truncating any higher-order bits. The executable is disassembled and the bytes making up each instruction are coalesced together as one block. We subsequently make use of the *n-gram* model which assumes a Markov property, that is, token occurrences are influenced only by the $n-1$ tokens before it. To form the first n-gram, we concatenate mnemonics 0 to $n-1$; to form the second n-gram, we concatenate 1 to n and so on.

One disadvantage of using mnemonic n-grams is that some instruction sequences may be reordered without affecting the program semantics. For example, the following two instruction sequences are semantically identical and yet they will give different 3-grams.

```
mov  ebp, esp          mov  ebp, esp
sub  esp, 0x10         movl -0x4(ebp), 0x1
movl -0x4(ebp), 0x1    sub  esp, 0x10
```

An alternative to the n-gram model is to use n-perms [10]. The n-perm model does not take order into consideration, and is set-based rather than sequence-based. So in the above example, there will be only one n-perm that represents both sequences: `mov, movl, sub`. The trade-off in using n-perms, however, is that there are not as many n-perms as n-grams for the same n, and this might affect the accuracy. The instruction mnemonics and 1-, 2- and 3-grams and corresponding n-perms for bubblesort are shown in Figure 3.

```
80483c4:  push  ebp
80483c5:  mov   ebp, esp
80483c7:  sub   esp, 0x10
80483ca:  movl  -0x4(ebp), 0x1
80483d1:  jmp   804844b
80483d3:  movl  -0x4(ebp), 0x0
80483da:  movl  -0x8(ebp), 0x1
80483e1:  jmp   8048443
80483e3:  mov   eax, -0x8(ebp)
...
```

1-grams	`push, mov, sub, movl, jmp,...`
2-grams	`push mov, mov sub, sub movl, movl jmp, jmp movl,...`
3-grams	`push mov sub, mov sub movl, sub movl jmp, movl jmp movl, jmp movl movl,...`
1-perms	`push, mov, sub, movl, jmp,...`
2-perms	`push mov, mov sub, sub movl, movl jmp,...`
3-perms	`push mov sub, mov sub movl, sub movl jmp, movl jmp movl,...`

Fig. 3. Instruction mnemonics and 1-, 2-, and 3-grams and corresponding n-perms for bubblesort (bottom) based on the first six instructions (top).

V. Control Flow Sub-Graphs

The second type of abstraction considered was control flow. To construct the control flow graph (CFG), the basic blocks (BBs) and their flow targets are extracted from the disassembly. A BB is a continuous instruction sequence for which there are no intermediate jumps into or out of the sequence. Call instructions are one of the exceptions as they are not treated as an external jump and are assumed to return. Conditional instructions, such as `cmov` and `loop` are another exception as they are treated as not producing additional control flow edges. The BBs form the nodes in the graph and the flow targets are the directed edges between nodes.

We do not index the whole CFG. Instead we extract sub-graphs of size k, or k-graphs. Our approach is similar to the one adopted by Krügel et al. [11]. We first generate from the CFG a list of connected k-graphs. This is done by choosing each block as a starting node and traversing all possible valid edges beginning from that node until k nodes are encountered.

Next, each sub-graph is converted to a matrix of size k by k. The matrix is then reduced to its canonical form via a pre-computed matrix-to-matrix mapping. This mapping may be computed off-line via standard tools such as Nauty [13], or by brute force since k is small (3 to 7).

Each unique sub-graph corresponds to a k^2-bit number. For $k = 4$, we obtain a 16-bit value for each sub-graph. Figure 4 shows the CFG of bubblesort, two k-graphs, `1-2-3-5` and `5-6-7-8`, and their canonical matrix forms `0x1214` and `0x1286` respectively. The canonical form in this example is the node labelling that results in the smallest possible numerical value.

One shortcoming of using k-graphs is that for small values of k, the uniqueness of the graph is low. For instance, if considering 3-graphs in the CFG of bubblesort, graphs `1-2-4`, `1-2-3`, `3-5-6` all produce an identical k-graph. To deal with this issue, we propose an extension to the k-graph which we call *extended* k-graphs. In addition to the edges solely between internal nodes, an extended k-graph includes edges that have one end point at an internal node, but have another at an external virtual node, written as V^*. This adds a row and a column to our adjacency matrix. The additional row contains edges that arrive from an external node; the extra column indicates edges with an external node as its destination. This allows us to now differentiate between the 3-graphs that we mentioned before.

	1	2	3
1	0	1	0
2	0	0	1
3	0	0	0

	1	2	3	V^*
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
V^*	0	1	0	0

	3	5	6
3	0	1	0
5	0	0	1
6	0	0	0

	3	5	6	V^*
3	0	1	0	0
5	0	0	1	0
6	0	0	0	1
V^*	1	1	0	0

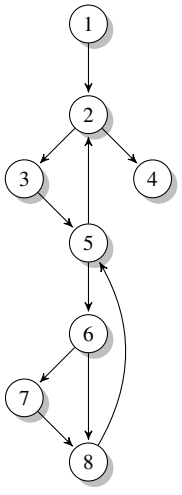
Fig. 5. Differentiating bubblesort's 3-graphs `1-2-3` and `3-5-6` with extended k-graphs. The left column shows the adjacency matrices for the k-graphs; the right column shows the corresponding extended k-graphs. The V^* node represents all nodes external to the sub-graph.

VI. Data Constants

The motivation for using constants is the empirical observation that constants do not change with the compiler or compiler optimisation. We considered two types of constants– 32-bit integers, and strings. We included integers used in computation, as well as integers used as pointer offsets; the strings we considered were ANSI single-byte null-terminated strings.

Our extraction algorithm is as follows: All constants are first extracted from an instruction. These include operands and pointer offsets. We explicitly exclude offsets associated with the stack and frame pointers, or `esp` and `ebp` as these depend on the stack layout and hence vary with the compiler.

The constants are then separated by type. Since we are considering a 32-bit instruction set the data could either be a 32-bit integer or a pointer. An address look up is made to ascertain whether the value v corresponds to a valid address in the data or code segment, and if so the data d_v is retrieved. Since d_v can also be an address, this process can continue recursively by another address look up on d_v. However, we do not do a second look up but stop at the first level of indirection. If d_v is a valid ANSI string, that is with valid ASCII characters and terminated by a null byte, we assign it as type string, otherwise, we do not use d_v. In all other cases, v is treated as an integer. Figure 6 shows the constants of the usage function of the `vdir` program compiled with `gcc` default options.



	2	5	3	1
2	0	0	1	0
5	1	0	0	0
3	0	1	0	0
1	1	0	0	0

	6	8	7	5
6	0	1	1	0
8	0	0	0	1
7	0	1	0	0
5	1	0	0	0

Fig. 4. CFG of bubblesort, two k-graphs, $k = 4$, and their canonical matrix ordering. The first sub-graph, `1-2-3-5`, corresponds to 0x1214, and the second, `5-6-7-8`, to 0x1286.

```
804eab5:   movl   0x8(esp),0x5
804eabd:   movl   0x4(esp),0x805b8bc
804eac5:   movl   (esp),0
804eacc:   call   804944c <dcgettext@plt>
...
805b8bc:   "Try `%s --help'..."
...

Constants      0x5, 0x0, "Try `%s --help'..."
```

Fig. 6. Data constants for a code snippet from the `usage` function of `vdir`.

VII. What Makes a Good Model?

How do we know when we have a good statistical model? Given a corpus of executables and a query, a model is one with high *precision* and *recall*. A true positive (*tp*) refers to a correctly retrieved document relevant to the query; a true negative (*tn*) is a correctly omitted irrelevant document; a false positive (*fp*) is an incorrectly retrieved irrelevant document; and a false negative (*fn*) is a missing but relevant document. The precision and recall are defined as

$$precision = \frac{tp}{tp + fp}$$
$$recall = \frac{tp}{tp + fn}$$

In other words, a good model will retrieve many relevant documents, omitting many other irrelevant ones. A measure that combines both precision and recall is the F measure, which comprises their harmonic mean.

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

This is also known as the F_1 measure and both precision and recall are weighted equally. For our purposes, we are interested in the F_2 measure, defined as the following.

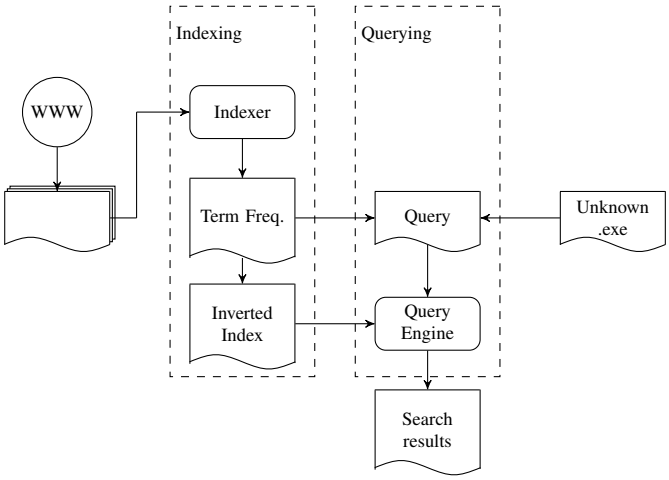


Fig. 7. Setup for indexing and querying.

$$F_2 = \frac{5 \cdot (precision \cdot recall)}{(4 \cdot precision + recall)}$$

The reason that we use the F_2 measure and not the F_1 measure is we want to retrieve as many relevant documents as possible and are less concerned about false positives. Thus recall is of higher priority than precision, and F_2 weights recall twice as much as precision.

VIII. Indexing and Querying

What we have discussed so far covers the extraction of terms from an executable. In this section, we describe how the tokens are incorporated into a standard text-based index, and how queries are made against this index that lead to meaningful search results.

Figure 7 shows a summary of the indexing and querying process. Since there are 52 different symbols, we can encode

a 32-bit integer as a 6-letter word for an alphabetic indexer. The indexing process is a straight-forward one—the corpus of binaries retrieved from the web is first processed to give a global set of terms S_{global}. The terms are processed by an indexer which produces two data mappings. The first is the term frequency mapping which maps a term to its frequency in the index; the second is the inverted index which maps a term to the list of documents containing that term.

We considered two query models—the Boolean model (BM), and the vector space model (VSM). BM is a set-based model and the document weights are assigned 1 if the term occurs in the document, or 0 otherwise. Boolean queries are formed by combining terms with Boolean operators such as `AND`, `OR` and `NOT`. VSM is distance-based and two documents are similar if the inner product of their weight vectors is small. The weight vectors are computed via the normalised term frequencies of all terms in the documents. Our model is based on the combination of the two: documents are first filtered via the BM, then ranked and scored by the VSM.

Given an executable of interest, we first decompose it into a set of terms in binary form. The binary terms are converted into strings of alphabetic symbols to give a set of terms S. For example, the mnemonic sequence `push, mov, push, push, sub` corresponds to the 4-grams `0x73f97373`, `0xf97373b3`, which encodes as the query terms `XvxFGF`, `baNUAL`.

A boolean expression Q is then constructed from the set of terms S. Unlike a typical user-entered text query, our problem is that the length of Q may be of the order of thousands of terms long, or, conversely, too short as to be too common. We employ three strategies which deal with these two issues, namely *term de-duplication*, *padding* and *unique term selection*.

Term de-duplication is an obvious strategy that reduces the term count of the query. For a desired query length l_Q, we select the first term t_0 and remove other occurrences of t_0 up to length $2 \cdot l_Q$. This process is repeated until we reach l_Q terms.

One problem which may arise is if Q is too short it may result in too many matches. To deal with this issue we add terms of high frequency that are not in S, negated with the logical `NOT`. For example, if l_Q is 3, and our query has two terms, `A` and `B`, we add the third term `NOT C`, where `C` is the term with the highest term frequency in S_{global}. This eliminates matches that may contain Q plus other common terms.

At the end of these steps, we obtain a bag of terms for each term category. We first construct the most restrictive query by concatenating the terms with `AND`, e.g. `XvxFGF AND baNUAL`, and the query is sent to the query engine which in turn queries the index and returns the results in the form of a ranked list. If this query returns no results, we proceed to construct a second query with unique term selection.

The aim of unique term selection is to choose terms with low document frequency, or rare terms. This is done if the size of S is larger than the maximum query length, l_Q. We control the document frequency threshold, $df_{threshold}$, which determines which terms in S to include in the query. Only terms whose frequency is below $df_{threshold}$ will be included. If $df_{threshold}$ is set too low, not enough terms will make it through, and conversely if $df_{threshold}$ is set too high, too many will be included in Q. The resulting terms are then concatenated with `OR` to form Q, e.g. `XvxFGF OR baNUAL`.

The rationale for using `AND` first is so that the query engine will find an exact match if one exists and return that one result purely based on BM. The second `OR` query is to deal with situations where an exact match does not exist, and we rely on VSM to locate the closest match.

Search results are ranked according to the default scoring formula used by the open source CLucene text search engine. Given a query Q and a document D, the similarity score function is defined as the following.

$$\mathit{Score}(Q, D) = \mathit{coord}(Q, D) \cdot C \cdot \frac{V(Q) \cdot V(D)}{|V(Q)|}$$

where *coord* is a score factor based on the fraction of all query terms that a document contains, C is a normalisation factor, $V(Q) \cdot V(D)$ is the dot product of the weighted vectors, and $|V(Q)|$ is the Euclidean norm. The $|V(D)|$ term is not used on its own as removing document length information affects the performance. Instead, a different document length normalisation factor is used. In our equation this factor is incorporated into C. Other boost terms have been omitted for brevity [14].

IX. Implementation

The tasks of disassembly, extracting n-grams, n-perms, control flow k-graphs, extended k-graphs and data constants were performed using the open source dynamic instrumentation library Dyninst version 8.0. The Nauty graph library [13] was used to convert k-graphs to their canonical form. The tasks of indexing and querying were performed using the open source text search engine CLucene 2.3.3.4. The term frequency map which was used in unique term selection was implemented as a Bloom filter [15] since it is a test of membership. In other words, the Bloom filter consisted of all terms below $df_{threshold}$. A total of 10,500 lines of C++ were written for the implementation of disassembly and code abstraction, and 1,000 lines of C++ for indexing and querying.

X. Evaluation

This section describes the experiments we conducted to answer the following questions.

- What is the optimal value of $df_{threshold}$?
- What is the accuracy of the various code abstractions?
- What is the effect of compiler optimisation on accuracy?
- What is the effect of the compiler on accuracy?
- What is the efficiency of binary code indexing and querying?

The first data set that we used for our evaluation were 2706 functions from the GNU C library version 2.16 comprising

1.18 million lines of code. The functions were obtained by compiling the suite under GCC -O1 and -O2 options, or *GCC1* and *GCC2* respectively. We excluded all initialisation and finalisation functions generated by the compiler, and had names such as `_init`, `_fini` and `__i686.get_pc_thunk.bx`. All experiments on this set, which we refer to as the *glibc* set, were conducted through two experiments. In the first experiment we indexed GCC1 and queried this index using GCC1 and GCC2. We then repeated this procedure with GCC2 as index and summed up the total over both experiments to obtain the precision, recall and F_2 measures.

The second data set was the coreutils 6.10 suite of tools, or the *coreutils* set, which we compiled under `gcc` and `clang` default configurations. This data set contained 98 binaries, 1205 functions, comprising 78,000 lines of code. To obtain the precision, recall and F_2 values we similarly indexed one set of functions and queried with the other as with the *glibc* set. All experiments were carried out on a Intel Core 2 Duo machine running Ubuntu 12.04 with 1 Gb of RAM.

Our results at a glance is summarised in Table II.

TABLE II
RESULTS AT A GLANCE.

Model	*glibc* F_2	*coreutils* F_2
Best n-gram (4-gram)	0.764	0.665
Best k-graph (5-graph)	0.706	0.627
Constants	0.681	0.772
Best mixed n-gram (1+4-gram)	0.777	0.671
Best mixed k-graph (5+7-graph)	0.768	0.657
Best composite (4-gram/5-graph/constants)	0.867	0.830

A. Optimal $df_{threshold}$

Recall that our query model is based on BM first, then ranked by VSM. However, we can further influence the input terms to the BM by varying $df_{threshold}$ which determines which terms to include in the query.

To investigate the optimal value of $df_{threshold}$, we ran an experiment using *glibc* and 4-grams as our term type. We then varied $df_{threshold}$ and measured the performance of the search results. We restricted the ranked list to at most 5 results, that is, if the correct match occurred lower than the top 5 results, it was treated as a false negative. Table III shows the results of this experiment. For example, $df_{threshold} \leq 1$ means that only terms having a document frequency less than or equal to 1 were included, and $df_{threshold} \leq \infty$ means that all terms were included.

Although there is an initial increase from $df_{threshold} \leq 1$ to $df_{threshold} \leq 2$, this was not sustained by increasing the value of $df_{threshold}$ further, and F_2 changes were insignificant. Since there was no gain in varying $df_{threshold}$, we decided to fix $df_{threshold}$ at ∞ throughout our experiments.

B. Comparison of N-Grams Versus N-Perms

Firstly, we compared the accuracy of n-grams with n-perms of instruction mnemonics, with n taking values from 1 to 4.

TABLE III
PERFORMANCE USING VARIOUS VALUES OF $df_{threshold}$ ON THE *glibc* SET USING 4-GRAMS.

$\leq df_{threshold}$	Precision	Recall	F_2
1	0.202	0.395	0.331
2	0.177	0.587	0.401
3	0.165	0.649	0.410
4	0.161	0.677	0.413
5	0.157	0.673	0.406
6	0.160	0.702	0.418
7	0.159	0.709	0.419
8	0.157	0.708	0.415
9	0.157	0.716	0.418
10	0.155	0.712	0.414
11	0.151	0.696	0.405
12	0.152	0.702	0.408
13	0.153	0.705	0.410
∞	0.151	0.709	0.408

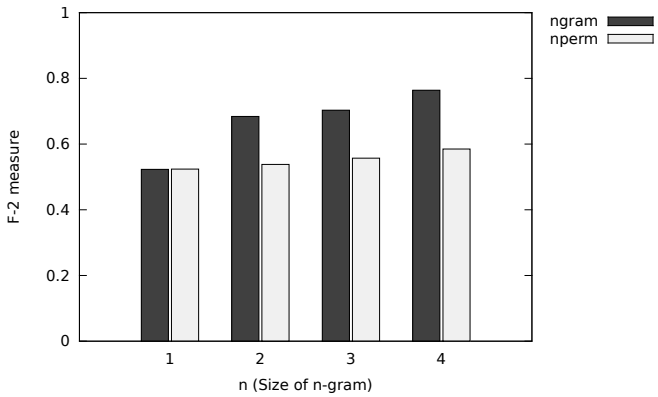


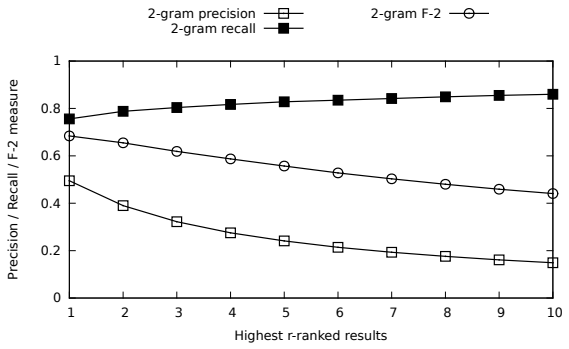
Fig. 8. The F_2 measures for n-grams and n-perms (glibc data set).

We did not consider values of n larger than 4. We used the *glibc* test set to ascertain the accuracy of these two methods in the presence of compiler optimisations. The results are shown in Figure 8. The overall best F_2 measure was 0.764, and was obtained using the 4-gram model. Both 1-gram and 1-perm models were identical as to be expected, but the n-gram model out-performed n-perms for $n > 1$. One explanation for this difference was that the n-perm model was too normalising so that irrelevant results affected recall rates, and this is evident looking at the number of unique terms generated by the two models (Table IV). The 2-gram model generated 1483 unique terms whilst the 4-perm model generated only 889.

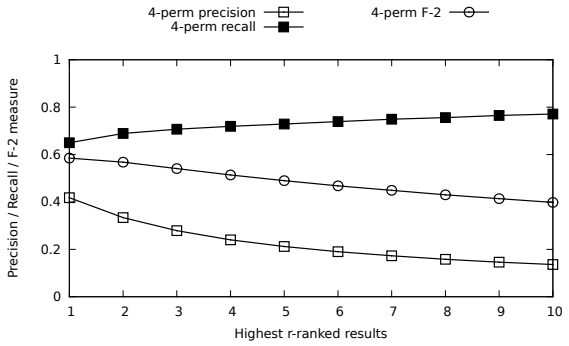
We proceeded to analyse 2-grams and 4-perms in more detail. We varied the precision and recall rates by adjusting

TABLE IV
THE NUMBER OF UNIQUE TERMS FOR n-GRAMS AND n-PERMS (*glibc* DATA SET).

n	n-gram	n-perm
1	121	121
2	1483	306
3	6337	542
4	16584	889



(a) 2-gram



(b) 4-perm

Fig. 9. The precision, recall rates and the F_2 measures for 2-grams and 2-perms of instruction mnemonics (*glibc* data set).

the threshold of the r highest ranked results obtained. For example, if this threshold was 1, we only considered the highest ranked result returned by the query engine and ignored the rest. The value of r was varied from 1 to 10. Figures 9a and 9b show that the maximum F_2 measure obtained for 2-grams was 0.684 at $r = 1$, and the precision and recall rates were 0.495 and 0.756 respectively. The corresponding maximum F_2 value was 0.585 for 4-perms also at $r = 1$.

We observe that there is a tension between precision and recall. As r increases the number of successful matches increases causing the recall to improve, but this also causes the false positives to increase, reducing precision.

The second larger *coreutils* data set was similarly tested with the n-gram and n-perm models, with $n = 1, 2, 3, 4$. Similar observations were made—n-grams out-performed n-perms for all n.

C. Mixed N-Gram Models

We next looked at mixed n-gram models to see if combining n-grams produced better performance. If we consider combining 1-gram to 4-gram models, there are a total of 6 possible paired permutations. These combined models were tested on the *glibc* and *coreutils* set and the results are shown in Table V. The two highest scores were obtained using 1- and 4-grams (1+4-gram) and 2- and 4-grams (2+4-gram) for the two data sets. This was surprising since 1-grams generated a small fraction of terms, e.g. 121, compared to 4-grams, e.g. 16584. Also notable was the fact that almost all mixed n-gram models performed better than the single n-gram models.

TABLE V
PERFORMANCE OF MIXED n-GRAM MODELS BY F_2 MEASURE.

	glibc	coreutils
1+2-gram	0.682	0.619
1+3-gram	0.741	0.649
1+4-gram	**0.777**	**0.671**
2+3-gram	0.737	0.655
2+4-gram	**0.777**	**0.675**
3+4-gram	0.765	0.671

TABLE VI
RESULTS FOR k-GRAPHS AND EXTENDED k-GRAPHS.

	glibc					
	k-graph			extended k-graph		
	Precision	Recall	F_2	Precision	Recall	F_2
3-graph	0.070	0.133	0.113	0.022	0.062	0.046
4-graph	0.436	0.652	0.593	0.231	0.398	0.348
5-graph	0.730	0.700	**0.706**	0.621	0.600	0.604
6-graph	0.732	0.620	0.639	0.682	0.622	**0.633**
7-graph	0.767	0.609	0.635	0.728	0.610	0.631
	coreutils					
	k-graph			extended k-graph		
	Precision	Recall	F_2	Precision	Recall	F_2
3-graph	0.110	0.200	0.172	0.042	0.080	0.068
4-graph	0.401	0.586	0.537	0.218	0.360	0.318
5-graph	0.643	0.623	**0.627**	0.553	0.531	0.535
6-graph	0.617	0.527	0.543	0.660	0.602	**0.613**
7-graph	0.664	0.560	0.578	0.663	0.566	0.583

D. Control Flow K-Graphs Versus Extended K-Graphs

In the next set of experiments, we evaluated the control flow k-graphs, and extended k-graphs for $k = 3, 4, 5, 6, 7$. The results are summarised in Table VI. The model which gave the highest F_2 was 5-graphs for the *glibc* data set at 0.706, and 5-graphs also for the *coreutils* data set at 0.627. This consistency was surprising given that there were thousands of different functions being considered.

The second observation was that the performance of extended k-graphs was lower than that of regular k-graphs. This difference was more marked for *glibc* than for *coreutils*, at 7 and 1.4 percentage points respectively. The implication is that extended k-graphs were in fact more normalising than k-graphs.

E. Mixed K-Graph Models

As with n-grams, we considered mixed k-graph models as a possible way to improve performance on single k-graph models. We limited mixed models to a combination of at most two k-graph models, giving us a total of 10 possibilities.

Again, the best mixed model was the same for both data sets. The 5+7-graph model gave the best F_2 value for both *glibc* (0.768) and *coreutils* (0.657) (Table VII). Lastly, the mixed k-graph models performed better than the single k-graph models.

TABLE VII
RESULTS FOR MIXED k-GRAPH MODELS.

	glibc F_2	*coreutils* F_2
3+4-graphs	0.607	0.509
3+5-graphs	0.720	0.630
3+6-graphs	0.661	0.568
3+7-graphs	0.655	0.559
4+5-graphs	0.740	0.624
4+6-graphs	0.741	0.624
4+7-graphs	0.749	0.649
5+6-graphs	0.752	0.650
5+7-graphs	**0.768**	**0.657**
6+7-graphs	0.720	0.624

TABLE VIII
RESULTS OF USING DATA CONSTANTS TO IDENTIFY FUNCTIONS IN THE *glibc* AND *coreutils* DATA SETS.

	Precision	Recall	F_2
glibc	0.690	0.679	0.681
coreutils	0.867	0.751	0.772

F. Data Constants

Table VIII shows the results of using the third type of code abstraction, data constants, to match functions compiled using different optimisations (*glibc*) and different compilers (*coreutils*). The performance was better for `coreutils` at 0.772 compared to `glibc` at 0.681. One possible explanation for this difference is the fact none of the functions in the *glibc* had strings, whilst 889 functions, or 40.3% of functions in the *coreutils* did. Also as a consequence of this, the number of functions having more than 110 terms was higher for *coreutils*—11.1% compared to only 5.3% in *glibc*.

G. Composite Models

Building upon the observation that mixed models were more successful than single models, the last set of models considered were composite ones, i.e. models that combined n-grams, k-graphs and constants. The terms from each individual model, for example 4-grams, were abstracted then concatenated to form a composite document for each function.

We considered two composite models for each data set. The first composite model was made up of the highest performing single model from each of the three categories; the second composite model was made up of the highest performing mixed model, except for constants. Thus, we tested the models comprising 4-gram/5-graph/constants and 1-gram/4-gram/3-graph/5-graph/constants for the *glibc* set. The corresponding models for the *coreutils* set were 4-gram/5-graph/constants and 2-gram/4-gram/5-graph/7-graph/constants. The results are given in Table IX.

Overall, the best composite model out-performed the best mixed models, giving an F_2 score of 0.867 and 0.830 for *glibc* and *coreutils* respectively. The highest scores for mixed models were 0.777 (1-gram/4-gram) and 0.772 (constants). One observation was that including more models did not necessarily result in better performance. This was evident from the fact that the composite models with 3 components fared better than the model with 5.

Instead of maximising F_2, we were also interested in the recall rates when the value of r, the number of ranked results, was 10, since we do not expect users of the search engine to venture beyond the first page of results. Considering only the top 10 ranked results, the recall rates were 0.925 and 0.878 respectively for *glibc* and *coreutils*.

Of the 342 false negatives from the *glibc* set, we found that 206 were small functions, having 6 instructions or less. Since Rendezvous uses a statistical model to analyse executable code, it is understandable that it has problems differentiating between small functions.

One of the largest functions in this group was `getfsent` from *glibc* (Figure 10). The output for `gcc -O1` and `gcc -O2`, or `getfsentO1` and `getfsentO2` respectively, differ significantly due to several factors. Firstly, the function `fstab_fetch` was inlined, causing the `mov` and `call` instructions in `getfsentO1` to be expanded to 8.

Secondly, there were two instruction substitutions: instruction `mov eax, 0x0` in `getfsentO1` was substituted by the `xor eax, eax` instruction which utilises 2 bytes instead of 5; the call to `fstab_convert` was substituted by an unconditional jump. In the latter substitution, the call was assumed to return, whereas the jump did not. This was evident from the fact that the stack was restored immediately prior to the jump. This altered the control flow graph since the edge from the `jmp` instruction to the final BB was no longer there in `getfsentO2`.

Thirdly, there were two occurrences of instruction reordering: The first being the swapping of the second and third instructions of both functions; the second was the swapping of the `test` and `mov` instructions following the call to `fstab_init`.

The sum of these changes resulted in the fact that there were no 3-grams, 4-grams nor data constants in common between the two functions, and the two 4-graphs did not match. In such cases, the matching could benefit from a more accurate form of analysis, such as symbolic execution [16], but this is left to future work.

TABLE IX
RESULTS OF THE COMPOSITE MODELS. WHERE INDICATED, VARIABLE r IS THE NUMBER OF RANKED RESULTS CONSIDERED, OTHERWISE $r = 1$.

		Precision	Recall	F_2
glibc	4-gram/5-graph/constants	0.870	0.866	**0.867**
	1-gram/4-gram/5-graph/7-graph/constants	0.850	0.841	0.843
	4-gram/5-graph/constants ($r = 10$)	0.118	**0.925**	0.390
coreutils	4-gram/5-graph/constants	0.835	0.829	**0.830**
	2-gram/4-gram/5-graph/7-graph/constants	0.833	0.798	0.805
	4-gram/5-graph/constants ($r = 10$)	0.203	**0.878**	0.527

```
struct fstab *getfsent( void ){
    struct fstab_state *state;
    state = fstab_init(0);

    if( state == NULL )
        return NULL;

    if( fstab_fetch(state) == NULL )
        return NULL;

    return fstab_convert(state);
}
```

Fig. 10. Source code of `getfsent`

TABLE X
AVERAGE AND WORST-CASE TIMINGS FOR *coreutils* SET.

		Average (s)	Worst (s)
Abstraction	n-gram	46.684	51.881
	k-graph	110.874	114.922
	constants	627.656	680.148
	null	11.013	15.135
Query construction		6.133	16.125
Query		116.101	118.005
Total (2410 functions)		907.448	981.081
Total per function		0.377	0.407

H. Timing

The final set of experiments was conducted to determine the time taken for a binary program to be disassembled, for the terms to be abstracted and for the query to return with the search results. We timed the *coreutils* set for this experiment, and included both the `gcc`-compiled code and the `clang`-compiled code to give a total of 2410 functions. Table X shows the average case as well as the worst-case timings for each individual phase. The "null" row indicates the time taken for Dyninst to complete the disassembly without performing any further abstraction. The total time was computed by summing the time taken by the three abstractions. Strictly speaking, our time is an overestimate since the binary was disassembled two more times than was necessary in practice. On the other hand, we did not do end-to-end timings to take into consideration the computational time required for the front-end system, so the timings are an approximation at best.

We found that a significant portion of time was spent in extracting constants from the disassembly. The reason is because the procedure is currently made up of several different tools and scripts, and we hope to streamline this procedure in future.

XI. DISCUSSION

A. Limitations

An important assumption made in this paper is that the binary code in question is not actively obfuscated. The presence of code packing, encryption or self-modifying code would make disassembly, and therefore code abstraction, difficult to perform. In practice, Rendezvous may require additional techniques, such as dynamic code instrumentation and symbolic execution, to analyse heavily obfuscated executables. However, as mentioned, we considered static analysis primarily for its efficiency. In future we hope to look at efficient ways to include dynamic methods in Rendezvous.

B. Threats to Validity

Threats to internal validity include the limited software tested, and limited number of compilers and compiler optimisation levels used. The good performance may be due to a limited sample size of software analysed, and future work will involve analysing larger code bases. It is possible that the `gcc` and `clang` compilers naturally produce similar binary code. Likewise, the output of `gcc -O1` and `gcc -O2` could be naturally similar.

The most important threat to external validity is the assumption that there is no active code obfuscation involved in producing the code under consideration. Code obfuscation, such as the use of code packing and encryption, may be common in actual binary code in order to reduce code size or to prevent reverse engineering and modification. Such techniques may increase the difficulty of disassembly and identification of function boundaries.

XII. RELATED WORK

The line of work that is most closely related to ours is that of binary clone detection. Sæbjørnsen et al. [17] worked on detecting "copied and pasted" code in Windows XP binaries and the Linux kernel by constructing and comparing vectors of features comprising instruction mnemonics, exact and normalised operands located within a set of windows in the code segment. The main goal was to find large code clones within the same code base using a single compiler and hence their method did not need to address issues with multiple compilers and their optimisations. In contrast, since the goal of Rendezvous is to do binary clone matching across different code bases, we needed to address the compiler optimisation problem and we believe our technique to be sufficiently accurate to be successful. Hemel et al. [18] looked purely at strings in the binary to uncover code violating the GNU public license. The advantage of their technique was that it eliminated the need to perform disassembly. However, as our experiments show, using only string constants we were only able to identify between 60 to 70% of functions. Other approaches include directed acyclic graphs [9], program dependence graphs [19] and program expression graphs [20]. We did not consider these approaches as the computational costs of these techniques are higher than what Rendezvous currently uses.

A closely related area is source code clone detection and search, and techniques may be divided into string-based, token-based, tree-based and semantics-based methods [21]. Examples include CCFinder [22], CP-Miner [23], MUDABlue [24], CLAN [25] and XIAO [26]. Rendezvous, however, is targeted at locating code in binary form, but borrows some inspiration from the token-based approach.

A related field is malware analysis and detection, whose goal is to classify a binary as being malicious, or belonging

to a previously known family. Code abstractions that have been studied include byte values [27], opcodes [28], control flow sub-graphs [11], call graphs [29], as well as run-time behavioural techniques [30]. Even though Rendezvous borrows techniques from this field, our aim is to do more fine grain analysis, and identify binary code at a function level. At the moment, we are only considering code obfuscation up to the level of the compiler and its different optimisations.

XIII. Conclusion

We present a prototype search engine called Rendezvous that exploits three independent analyses to identify known functions in a binary program. We have found conclusive evidence that combining these analyses allows us to identify code in a way that is robust against different compilers and optimisations, and achieve F_2 measures of 0.867 and 0.830 on two data sets. As an early prototype, Rendezvous takes 0.407s to analyse a function in the worst case. Future work will involve improving its efficiency and performing further evaluation.

XIV. Acknowledgements

We would like to thank Khilan Gudka for his comments on an earlier draft. We would like to also thank the anonymous reviewers for their constructive feedback. The first author acknowledges the support of DSO National Laboratories during his PhD studies.

References

[1] "Github," http://github.com/.

[2] A. Mockus, "Large-scale code reuse in open source software," in *Emerging Trends in FLOSS Research and Development, 2007. FLOSS '07. First International Workshop on*, may 2007, p. 7.

[3] H. Welte, "Current developments in GPL compliance," http://taipei.freedomhec.org/dlfile/gpl_compliance.pdf, 2012.

[4] "Google Code Search," http://code.google.com/codesearch.

[5] Ohloh, "The open source network," http://www.ohloh.net/.

[6] M. Mock, "Dynamic analysis from the bottom up," in *Proc. 1st ICSE Int. Workshop on Dynamic Analysis (WODA)*. IEEE C.S., 2003, pp. 13–16.

[7] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *ACSAC*, 2007, pp. 421–430.

[8] B. Schwarz, S. K. Debray, and G. R. Andrews, "Disassembly of executable code revisited," in *9th Working Conference on Reverse Engineering (WCRE 2002)*, A. van Deursen and E. Burd, Eds. IEEE Computer Society, 2002, pp. 45–54.

[9] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing*, ser. SAC '05. ACM, 2005, pp. 314–318.

[10] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1–2, pp. 13–23, 2005.

[11] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *RAID*, 2005, pp. 207–226.

[12] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, Nov. 2000.

[13] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[14] Apache Software Foundation, "Similarity (Lucene 3.6.2 API)," http://lucene.apache.org/core/3_6_2/api/core/org/apache/lucene/search/Similarity.html.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[16] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Combining symbolic execution with model checking to verify parallel numerical programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 10:1–10:34, May 2008.

[17] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. ACM, 2009, pp. 117–128.

[18] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th International Working Conference on Mining Software Repositories (MSR 2011)*, A. van Deursen, T. Xie, and T. Zimmermann, Eds. IEEE, 2011, pp. 63–72.

[19] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. ACM, 2006, pp. 872–881.

[20] C. Gautier, "Software plagiarism detection with PEGs," Master's thesis, University of Cambridge, 2011.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. IEEE Computer Society, 2007, pp. 96–105.

[22] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions on*, vol. 28, no. 7, pp. 654 – 670, jul 2002.

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

[24] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories," *J. Syst. Softw.*, vol. 79, no. 7, pp. 939–953, Jul. 2006.

[25] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 364–374.

[26] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "XIAO: tuning code clones at hands of engineers in practice," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 369–378.

[27] W.-J. Li, K. Wang, S. Stolfo, and B. Herzog, "Fileprints: identifying file types by n-gram analysis," in *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, june 2005, pp. 64 – 71.

[28] D. Bilar, "Opcodes as predictor for malware," *Int. J. Electron. Secur. Digit. Forensic*, vol. 1, no. 2, pp. 156–168, Jan. 2007.

[29] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Proceedings of Symposium sur la sécurité des technologies de l'information et des communications (SSTIC'05)*, 2005.

[30] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, 2009.