Technical Report

UCAM-CL-TR-844
ISSN 1476-2986

Number 844

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Decompilation as search

Wei Ming Khoo

November 2013

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted August 2013 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Hughes Hall.

ISSN 1476-2986

Abstract

Decompilation is the process of converting programs in a low-level representation, such as machine code, into high-level programs that are human readable, compilable and semantically equivalent. The current *de facto* approach to decompilation is largely modelled on compiler theory and only focusses on one or two of these desirable goals at a time.

This thesis makes the case that decompilation is more effectively accomplished through search. It is observed that software development is seldom a clean slate process and much software is available in public repositories. To back this claim, evidence is presented from three categories of software development: corporate software development, open source projects and malware creation. Evidence strongly suggests that code reuse is prevalent in all categories.

Two approaches to search-based decompilation are proposed. The first approach borrows inspiration from information retrieval, and constitutes the first contribution of this thesis. It uses instruction mnemonics, control-flow sub-graphs and data constants, which can be quickly extracted from a disassembly, and relies on the popular text search engine CLucene. The time taken to analyse a function is small enough to be practical and the technique achieves an F_2 measure of above 83.0% for two benchmarks.

The second approach and contribution of this thesis is perturbation analysis, which is able to differentiate between algorithms implementing the same functionality, e.g. bubblesort versus quicksort, and between different implementations of the same algorithm, e.g. quicksort from Wikipedia versus quicksort from Rosetta code. Test-based indexing (TBI) uses random testing to characterise the input-output behaviour of a function; perturbation-based indexing (PBI) is TBI with additional input-output behaviour obtained through perturbation analysis. TBI/PBI achieves an F_2 measure of 88.4% on five benchmarks involving different compilers and compiler options.

To perform perturbation analysis, function prototyping is needed, the standard way comprising liveness and reaching-definitions analysis. However, it is observed that in practice actual prototypes fall into one of a few possible categories, enabling the type system to be simplified considerably. The third and final contribution is an approach to prototype recovery that follows the principle of conformant execution, in the form of inlined data source tracking, to infer arrays, pointer-to-pointers and recursive data structures.

Acknowledgments

I gratefully acknowledge the financial support of DSO National Laboratories, which made it possible for me to pursue this PhD. I thank Tan Yang Meng and Chia Hock Teck, who first encouraged me to take this journey; Ross Anderson, for deciding to take a green and clueless individual under his wing as a PhD student in 2009; Alan Mycroft, for agreeing to co-advise this same green and clueless PhD student; Hassen Saïdi, for having me as a summer intern and for getting me started on the fascinating topic of decompilation.

I thank my co-authors and collaborators Pietro Lió, Hyoungshick Kim, Michael Meeks and Ed Robbins, for fruitful discussions that have given me new perspectives and have helped me grow as a researcher. For providing me with much-needed advice, help and encouragement along the way, I thank Saad Aloteibi, Jonathan Anderson, Joseph Bonneau, Omar Choudary, Richard Clayton, Saar Drimer, Christopher Gautier, Khilan Gudka, Markus Kuhn, Steven Murdoch, Frank Stajano, Raoul Urma, Robert Watson and Rubin Xu. I am grateful to Graham Titmus and Chris Hadley for help with setting up the Rendezvous server, and Laurent Simon for help with server testing. I am thankful to Andrew Bernat and Frank Eigler for help with the Dyninst API and the Dyninst team for creating a great tool.

Thank you, Dad, Mum, *Pa*, *Mi* and Li Ying, for always being there, especially during times when I needed the most support and encouragement.

To Grace, Natalie and Daniel: thank you for being my inspiration. You are a gift, and my joy.

To my wife, Shea Lin: thank you for your steadfast support when I was running long experiments, writing or away, for being my comfort when the going was tough, and for your uncanny ability to see past setbacks. Thank you for being you.

Soli Deo gloria

Contents

1

Introduction

"You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.)"

– Ken Thompson, *Reflections on Trusting Trust*, 1984

Possibly the most well-known software backdoor was described by Ken Thompson in his famous 1984 ACM Turing award acceptance speech he titled *Reflections on Trusting Trust* [1]. A backdoor is a metaphor for a feature in a piece of software that exhibits behaviour that is more permissive than normal operations would allow, and is well-hidden. The backdoor that Thompson wrote existed in the `login` program on a UNIX system, and allowed anyone with knowledge of a backdoor password to log in as, or gain access to, any user account on that machine. From a software auditor's perspective, the problem with this backdoor was that it was not visible by inspecting the source code of the `login` program. This was because while the backdoor existed in the executable, it was inserted during compilation by a specially crafted C compiler.

In addition, this unique compiler functionality was not visible by inspecting the source code of the compiler either because the compiler executable was modified such that it could recognise that it was compiling *itself* and therefore insert the backdoor-inserting ability into its compiled self. Thus, this backdoor could not even be spotted by inspecting the source code of the compiler.

One method to detect a trojanised compiler was proposed by David Wheeler in his PhD dissertation using an approach called *diverse double-compiling* [2]. The key idea is to make use of the fact that source code compiled by two different but correct compilers must necessarily be functionally equivalent. Therefore, the solution is to do compilation twice on the compiler source code using the suspicious compiler and a second imported compiler. If both compilers are correct, the compilation will yield two compilers which are functionally equivalent. To test that they are, a second compilation by these two compiled compilers will yield a third and fourth compiler that is equal byte-for-byte under certain assumptions. If they are not equal, there is a difference in functionality between the two compilers that were initially used (see Figure 1.1).

Another approach to detect a trojanised compiler is to use *software reverse engineering*. We can use a disassembler, such as the UNIX utility `objdump`, to inspect the machine instructions contained within an executable. Knowing this, however, the `objdump` program could also be compromised with a backdoor that selected parts of the `login` executable

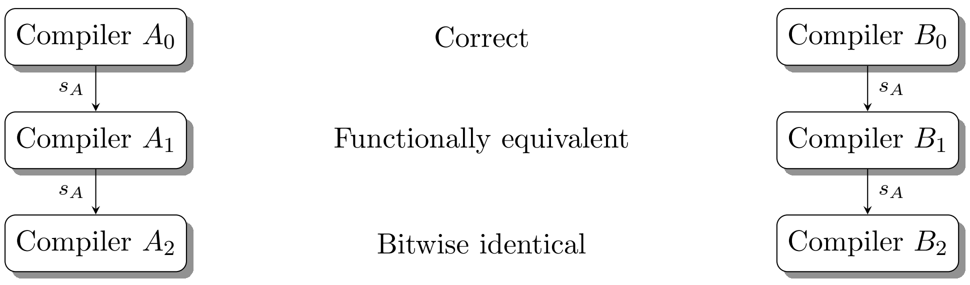


Figure 1.1: Overview of the diverse double-compiling approach to detect a trojanised compiler. Compilers A_0 and B_0 are assumed to be two correct compilers. Using A_0 and B_0 to compile the same compiler source, s_A, gives rise to compilers A_1 and B_1 respectively, which are expected to be functionally equivalent. A second iteration of this process produces compilers A_2 and B_2, which are expected to be bitwise identical, provided s_A is deterministic. Otherwise, the initial assumption is wrong and one, or both, of the initial compilers contains an error.

and C compiler to display. We cannot trust the disassembler in the operating system, but we can similarly import one or more external disassemblers. The chance of having the exact same backdoor in n independently developed disassemblers is small for a sufficiently large n. When run on the compiler or `login` executable, all disassemblers should give the same sequence of machine instructions, otherwise there is something wrong and we can examine the difference between the disassemblies.

In January of 2010, the world was introduced to the first-ever targeted cyberweapon known as *Stuxnet*. Stuxnet was a computer program designed specifically to hunt down a particular industrial control system known as the Siemens Simatic WinCC Step7 controller, but not just any Step7 system! Most of the Stuxnet infections were located in Iran, a country that was never known for huge malware outbreaks [3].

The one characteristic that separated Stuxnet from other malware was its reliability. Stuxnet had 8 methods of replicating itself from machine to machine, and it contained 5 exploits that had 100% reliable code execution capability on Windows Vista and later. An exploit is similar to a backdoor in that it allows for more permissive behaviour by the entity using it. The subtle difference is that a backdoor is usually intentional on the part of the software designer, while an exploit is not. Stuxnet was reliable to the point of being able to defeat anti-exploit security mechanisms such as data execution prevention (DEP), circumvent 10 different security products from companies such as Microsoft, Symantec and McAfee, and fool host-based intrusion detection systems by using a customised dynamic-linked library (DLL) loading procedure. Its main kernel driver was digitally signed by a valid code signing certificate so that it could be installed on any Windows operating system since Windows 2000 without any users being notified [4].

The goal of Stuxnet was to cause damage to a specific group of centrifuges used in Iranian uranium enrichment, and it did this by first seeking out computers containing Step7 software that controlled a series of 33 or more frequency converter drives which in turn controlled an array of exactly 164 spinning centrifuges. If this specific configuration was not found on a computer, it would simply shut itself down quietly. However, if it found what it was looking for, Stuxnet would take over the controller, and periodically, once every 27 days, it would speed up and slow down the spinning frequency of the centrifuges for about an hour each time before restoring it to the normal frequency. Assuming that

Stuxnet was launched in June 2009, it took about a year for it to be discovered, and a further 5 months for software analysts and industrial control systems experts to decipher the true intent of Stuxnet's payload. Stuxnet was possibly the most complex "malware" discovered to-date.

Analysing a program for the presence of a trojan or a malware sample is a task known as software reverse engineering. The goal of reverse engineering (RE) is to reconstruct a system's design from the system's details and extract the intention from the implementation, and it applies as much to software as it does to any engineered system. Software RE is important for tasks such as vulnerability analysis, malware analysis and identifying software copyright infringement. Vulnerability analysis is the process of looking for intended or unintended features in a piece of software or a system that would allow exceptionally permissive behaviour if exercised. As Balakrishnan and Reps pointed out, "what you see [in the source code] is not what you execute [5]". Malware analysis is the process of understanding the capabilities of a malicious program, and/or being able to uniquely identify future versions of it. Software RE can also be used to investigate verbatim copying of proprietary or protected code, complementing source code auditing, especially if the software uses third-party code to which the publisher does not have source code access, or if obtaining the source code is challenging [6].

1.1 Decompilation

The goals of decompilation, producing compilable, semantically equivalent and readable code, are difficult to achieve, even with today's tools.

The standard approach to software reverse engineering is *decompilation*, also known as reverse compilation. The goal of Hex-Rays, a commercial decompiler, is to convert "executable programs into a human readable C-like pseudocode text" [7]; the Boomerang decompiler aims "to create a high level, compilable, possibly even maintainable source file that does the same thing [as the executable]" [8]. Thus, the goals of decompilation are three-fold.

Compilable A decompiler should produce a program that can be re-compiled to an executable.

Semantically equivalent A decompiled program should do the same thing as the original executable.

Readable The decompiled output should be comprehensible, for example, with well-defined functions, variables, control-flow structure, high-level expressions and data type information, which would not otherwise exist.

Table 1.1 compares the output of three decompilers: free but closed source Reverse Engineering Compiler (REC) Studio version 4, open source Boomerang 0.3.1 and commercial Hex-Rays version 6.2. The decompilers were tested on three functions, the Fibonacci function, bubblesort and the GNU C library udivmoddi4 function, and assessed based on compilability, equivalence of the output and readability. The function *udivmoddi4* given unsigned integers a and b computes $a \mod b$ and a/b. The outputs of the three functions are listed in Figures 1.2, 1.3 and 1.4 for comparison. We use the short-hand, e.g. *Fibonacci-Boomerang*, to refer to the decompiled output of the target program by a decompiler, e.g. the decompiled output of the Fibonacci executable by Boomerang.

	REC Studio 4	Boomerang 0.3.1	Hex-Rays 6.2
Compilable	no (undefined macros)	some (with some manual editing)	yes
Equivalence	some	some	some
Readability			
Variable types/ names	no (unknown types)	yes	some
Control-flow	yes	yes	yes
Higher-order expressions	no (assembly present)	yes	no (assembly present)
Total	1.5/5	4/5	3/5

Table 1.1: Comparison of the output of three currently available decompilers.

With respect to producing compilable output, REC used undeclared macros `_pop()` and `_push()` and internal types `_unknown_` and `intOrPtr`, which made the C output uncompilable. It is not known why REC typed variable `_v16` in function `L00401090` of bubblesort a `char`, which was incompatible with type `intOrPtr`, the first parameter type of `L00401000`. This type mismatch would be flagged by the compiler as an error. *Bubblesort-Boomerang* was compilable with some manual editing, such as replacing the internal type `__size32` to `int`, and declaring a few overlooked variables. However, the first parameter in the main function of *bubblesort-Boomerang*, `&0`, was not a valid C expression (Listing 1.7).

In terms of equivalence, all three decompilers produced an equivalent C program for Fibonacci. For the bubblesort program, however, none of the decompilers correctly inferred the array in `main`, choosing instead to type the three-element array as three integers (Hex-Rays), two integers and a character (REC), or a constant (Boomerang). Indeed, only the first array element is read in `main`; the second and third array elements are only read within `bubblesort`. Without array bounds information, it is challenging to deduce the array in `main`. Hex-Rays comes the closest to decompiling an equivalent bubblesort C program, inferring the three-element array as three integers. However, the equivalence of the recompiled program to the original bubblesort program depends on the order the compiler chooses to arrange the integers on the stack. The Hex-Rays-decompiled variable order was `v1, v2, v3`, i.e. `v1` at the lowest stack address. When the *bubblesort-HexRays* C program was recompiled with `gcc`, the disassembly revealed that `gcc` had chosen to swap the locations of `v2` and `v3` on the stack, resulting in a different variable order, `v1, v3, v2`, and thus a semantically different bubblesort program from the original (Listing 1.8). For the udivmoddi4 function, the `if (d1 != 0)` statement in line `u5` (Listing 1.9) and the `if (param2 != 0)` statement in line `ub2` (*udivmoddi4-Boomerang*, Listing 1.11) were not equivalent since the variable `d1`, which comprised the most significant 32 bits of the 64-bit parameter `d` (line `u3`), did not equate to `param2`.

In terms of readability, Boomerang was arguably the most readable, followed closely by Hex-Rays. Boomerang was the only decompiler to not rely on macros, choosing instead to inline all low-level operations. REC was the only decompiler to leave the types as `intOrPtr` and `_unknown_`, possibly due to the lack of type information, while the other two chose to use integers. None of the decompilers managed to reconstruct the high-level expression `fib(x - 1) + fib(x - 2)`, however the order of the two calls were preserved. Hex-Rays was unable to infer the first Bubblesort parameter as a pointer, instead

Listing 1.1: Original

```
int fib ( int x ) {
   if ( x > 1 )
      return fib(x - 1) + fib(x - 2);
   else
      return x;
}
```

Listing 1.2: REC

```
L00401000( intOrPtr _a4 ) {
   _unknown_ __ebp;
   _unknown_ _t8;

   if (_a4 <= 1 ) {
      return _a4;
   }
   _t8 = L00401000(_a4 - 1);
   return L00401000(_a4 - 2) + _t8;
}
```

Listing 1.3: Boomerang

```
int fib ( int param1 ) {
   int eax; // r24
   int eax_1; // r24{30}

   if ( param1 <= 1 ) {
      eax = param1;
   } else {
      eax = fib(param1 - 1);
      eax_1 = fib(param1 - 2);
      eax = eax_1 + eax;
   }
   return eax;
}
```

Listing 1.4: Hex-Rays

```
int __cdecl fib ( int a1 ) {
   int v1; // esi@2
   int result; // eax@2

   if ( a1 <= 1 ) {
      result = a1;
   } else {
      v1 = fib(a1 - 1);
      result = v1 + fib(a1 - 2);
   }
   return result;
}
```

Figure 1.2: Listings for **1.1** the Original Fibonacci program, and output of the three decompilers **1.2** REC 4, **1.3** Boomerang 0.3.1 and **1.4** Hex-Rays 6.2, edited for length.

choosing to leave it as an integer. Considering the decompiled udivmoddi4 C programs, *udivmoddi4-HexRays* was in fact shorter than the equivalent fragment of the original udivmoddi4 function—the `BSR` (bit scan forward) instruction was left as is, as was the expression `HIDWORD(a3)`, which corresponded to register `EDX` (Listing 1.10). None of the decompilers recovered the correct prototype for udivmoddi4, which was `UDWtype udivmoddi4(UDWtype, UDWtype, UDWtype *)` or the equivalent `unsigned long long udivmoddi4(unsigned long long, unsigned long long, unsigned long long *)`. Hex-Rays inferred `int udivmoddi4(unsigned int, unsigned int, __int64, int)`, choosing to split the first 64-bit parameter into two 32-bit integers, inferring the second as `signed int64` instead of `unsigned int64` and inferring the third as an integer instead of a pointer; Boomerang was closer with `void udivmoddi4(unsigned long long, unsigned long long, unsigned long long *, unsigned long long, unsigned long long)`, adding two extra parameters. No output was obtained from REC for function udivmoddi4.

Overall, based on the limited programs tested, the focus of the three decompilers appeared to be largely on producing readable C-like code; equivalence was less of a priority. With the exception of Hex-Rays, manual editing was inevitable if compilable code was a requirement.

Today, the *de facto* decompilation approach involves several steps—disassembly [9], control-flow reconstruction [10], data-flow analysis, control-flow structuring, type inference, and code generation [11, 12, 13]. These are largely modelled on compiler theory. However, with the availability of numerous public code repositories today, such as Github.com, Google

Listing 1.5: Original

```
void bubblesort (int *a, int size) {
   int swapped = 1;
   while( swapped ) {
      int i;
      swapped = 0;
      for(i=1; i<size; i++) {
         if( a[i-1] > a[i] ) {
            int tmp;
            tmp = a[i-1];
            a[i-1] = a[i];
            a[i] = tmp;
            swapped = 1;
         }
      }
   }
}

int main(int argc, char **argv){
   int x[3]; x[0] = 0; x[1] = 10; x[2] = 20;
   bubblesort(x, 3);
   return 0;
}
```

Listing 1.6: REC

```
L00401000( intOrPtr _a4, intOrPtr _a8 ) {
   signed int _v8, _v12; intOrPtr _v16;
   _unknown_ __esi, __ebp; _v8 = 1;
   while(_v8 != 0) {
      _v8 = 0; _v12 = 1;
      while(_v12 < _a8) {
         if( *((intOrPtr*)(_a4 + _v12 * 4 - 4)) >
              *((intOrPtr*)(_a4 + _v12 * 4))) {
            _v16 = *((intOrPtr*)(_a4+_v12*4-4));
             *((intOrPtr*)(_a4 + _v12 * 4 - 4)) =
                  *((intOrPtr*)(_a4 + _v12 * 4));
             *((intOrPtr*)(_a4 + _v12 * 4)) = _v16;
            _v8 = 1;
         }
         _v12 = _v12 + 1;
      }
   }
}

L00401090() {
    intOrPtr _v8 = 20, _v12 = 10; char _v16 = 0;
    _unknown_ __ebp;
    L00401000( &_v16, 3);
    return 0;
}
```

Listing 1.7: Boomerang

```
void bubblesort(__size32 *param1, int param2) {
   unsigned int eax, eax_1;
   int ebx, ecx; __size32 esi;
   if (param2 > 1) {
      eax = 1; esi = 0;
      for(;;) {
         eax_1 = eax;
         ecx = *(param1+(eax_1-1)*4);
         ebx = *(param1 + eax_1 * 4);
         if (ecx > ebx) {
            *(int*)(param1+(eax_1-1)*4) = ebx;
            *(int*)(param1 + eax_1 * 4) = ecx;
            esi = 1;
         }
         eax = eax_1 + 1;
         if (eax_1 + 1 != param2) continue;
         if (esi == 0) break;
         eax = 1; esi = 0;
      }
   }
}

int main(int argc, char *argv[]) {
    bubblesort(&0, 3);
    return 0;
}
```

Listing 1.8: Hex-Rays

```
void __cdecl bubblesort(int a1, signed int a2) {
  signed int v2, i; int v4, v5;
  if ( a2 > 1 ) {
    v2 = 1;
    for ( i = 0; ; i = 0 ) {
      do {
        v4 = *(_DWORD *)(a1 + 4 * v2 - 4);
        v5 = *(_DWORD *)(a1 + 4 * v2);
        if ( v4 > v5 ) {
          *(_DWORD *)(a1 + 4 * v2 - 4) = v5;
          *(_DWORD *)(a1 + 4 * v2) = v4;
          i = 1;
        }
        ++v2;
      } while ( v2 != a2 );
      if ( !i ) break;
      v2 = 1;
    }
  }
}

int __cdecl main() {
  int v1 = 0, v2 = 10, v3 = 20;
  bubblesort((int)&v1, 3);
  return 0;
}
```

Figure 1.3: Listings for **1.5** the Original Bubblesort program, and output of the three decompilers **1.6** REC 4, **1.7** Boomerang 0.3.1 and **1.8** Hex-Rays 6.2, edited for length.

Code and Microsoft's CodePlex.com, this thesis proposes performing decompilation as search.

A similar shift occurred in machine translation. Translating between human languages is hard, and computer scientists tried for two generations to tackle it using natural-language progamming techniques that relied on syntactical analysis. Yet this problem is solved

Listing 1.9: Original

```
typedef unsigned long long UDWtype;
typedef long long DWtype;
typedef unsigned int UWtype;
typedef int Wtype;
struct DWstruct { Wtype low, high;};
typedef union { struct DWstruct s; DWtype ll; }
    DWunion;
...
static UDWtype
__udivmoddi4 (UDWtype n, UDWtype d, UDWtype *rp) {
u1: UWtype b, bm; DWunion nn, dd;
u2: nn.ll = n; dd.ll = d;
u3: UWtype d0 = dd.s.low; UWtype d1 = dd.s.high;
u4: UWtype n0 = nn.s.low; UWtype n1 = nn.s.high;
  ...
u5: if (d1 != 0) {
     if (d1 <= n1) {
        /* count_leading_zeros(count, x) counts the
             number of zero-bits from the
   msb to the first nonzero bit in the UWtype X. */
        count_leading_zeros (bm, d1);

        if (bm != 0) {
           UWtype m1, m0;
           /* Normalize. */
           b = W_TYPE_SIZE /* 32 */ - bm;
           d1 = (d1 << bm) | (d0 >> b);
  ...
}
```

Listing 1.10: Hex-Rays

```
signed int __usercall _udivmoddi4<eax>(unsigned
     int a1<eax>, unsigned int a2<edx>, __int64
     a3, int a4) {
  unsigned int v5 = a2, v16; char v25;
  _EDX = HIDWORD(a3);
  ...
  if ( HIDWORD(a3) ) {
    if ( HIDWORD(a3) <= v5 ) {
      __asm { bsr ecx, edx }
      v25 = _ECX ^ 0x1F;
      if ( _ECX ^ 0x1F ) {
        v16 = (HIDWORD(a3) << v25) | ((unsigned
             int)a3 >> (32 - v25));
  ...
}
```

Listing 1.11: Boomerang

```
void __udivmoddi4(unsigned long long param1, unsigned long long param2, unsigned long long *param3,
     unsigned long long param4, unsigned long long param5) {
ub1: unsigned long long ecx = param4, ecx_1, ecx_2, ebp_1;
  ...
ub2: if (param2 != 0) {
      if (param2 <= param5) {
         if (param2 != 0) {
            ecx = 32;
            do {
                ecx_1 = ecx; ecx = ecx_1 - 1;
            } while ((param2 >> ecx_1 - 1 & 0x1ffffff) == 0);
         }
         ecx_2 = ecx;
         if ((ecx_2 ^ 31) != 0) {
            ecx = (32 - (ecx_2 ^ 31));
            ebp_1 = param1 >> ecx | param2 << (ecx_2 ^ 31);
  ...
}
```

Figure 1.4: Listings for **1.9** a fragment of the original udivmoddi4, and corresponding output of **1.10** Hex-Rays 6.2 and **1.11** Boomerang 0.3.1, edited for length.

much more successfully nowadays by Google Translate[1]. According to Google, when a translation is generated, "it looks for patterns in hundreds of millions of documents to help decide on the best translation… . The more human-translated documents that Google Translate can analyse in a specific language, the better the translation quality will be[2]."

Instead of asking "How do we perform decompilation?" we want to ask "Given a candidate decompilation, how likely is it to be correct?". If a good enough match does exist, we get equivalence, compilable code, and readability in terms of variable names, comments

[1]http://translate.google.com
[2]http://translate.google.com/about

and labels which are challenging to obtain via current approaches. The more source code there is to index, the better the decompilation will be.

1.2 Chapter outline

The rest of this dissertation is structured in the following manner.

Models of decompilation: a survey (Chapter 2) surveys prior work in decompilation, tracing the five different models of decompilation in an area of research that spans more than fifty years.

Decompilation as search (Chapter 3) provides the motivation and the arguments for performing decompilation as a search process. It addresses the questions: "How prevalent is code reuse?" and "What components tend to be reused?". A study of code reuse in practice is provided, with evidence taken from: past research on code reuse in the software industry and in open source projects, software repositories on social coding site GitHub.com, leaked ZeuS malware source code and software copyright infringement lawsuits. This chapter concludes with a proposed research agenda for search-based decompilation.

Token-based code indexing (Chapter 4) contributes a code indexing technique that is focussed on speed, optimised for accuracy. It makes use of three code features at its core—instruction mnemonics, control-flow subgraphs and data constants—to perform binary matching. This work was published at the 10th Working Conference on Mining Software Repositories 2013 [14].

Perturbation analysis (Chapter 5) addresses the problem of differentiating between different algorithms implementing the same functionality, e.g. bubblesort versus quicksort, and between different implementations of the same algorithm, e.g. bubblesort from Wikipedia versus bubblesort from Rosetta code. This involves examining the input-output relationship of a function.

Prototype recovery via inlined data source tracking (Chapter 6) describes an approach to prototype recovery using data tracking and follows on from Chapter 5 by providing the basis with which to perform perturbation analysis. Unlike prior work, the focus is on discerning broad type categories—non-pointers, pointers, pointer-to-pointers—rather than the full range of C types.

Rendezvous: a prototype search engine (Chapter 7) is a demonstration of how a search-based decompiler might look like in practice. Sample screenshots are given as well as its results and performance.

Chapter 8 concludes by summarising the contributions made and discussing future work in search-based decompilation.

2

Models of decompilation: a survey

From an academic perspective, work on decompilation spans more than fifty years of research. As a nascent discipline in the 1960s, the earliest approaches to decompilation can perhaps be described as being *heuristics-driven*. The *compiler model*, which emerged in 1973, gathered considerable momentum and is currently considered to be the *de facto* approach to decompilation. The *formal methods model* is a more recent development, with publications first appearing in 1991. Although not considered to be decompilation by some, the usefulness of *assembly-to-C translation* has earned it a place among the other models, with papers being published from 1999. Finally, the *information-flow model*, proposed in 2008, views source code and machine code as high and low security information respectively and decompilation as exploiting information leakage to reconstruct an equivalent source language program for an observed machine language program. Both Cifuentes [11] and Van Emmerik [13] give comprehensive summaries of the history of decompilers for the periods up to 1994 and 2007 respectively. One example of a post-2007 development is the emergence of C++ decompilers [15]. What follows is the history of decompilation seen through the lens of the five aforementioned models of decompilation. When a system or methodology spans more than one model, the most representative one is chosen.

2.1 Heuristics-driven model

The heuristics-driven model exploits knowledge of patterns in executables to derive heuristics with which to perform decompilation, with the main drawback of being machine and/or compiler dependent.

The earliest decompilers were pattern-based, beginning in the 1960s with the Donelly-NELIAC (D-NELIAC) and Lockheed Missiles and Space Company (LMSC) NELIAC decompilers which used platform-specific rules to translate program instructions into NELIAC, a variant of the ALGOL-58 programming language. Platforms worked on included UNIVAC, CDC and IBM [16, 17]. Sassaman adopted a similar approach to translate IBM 7000 instructions into FORTRAN in 1966 [18].

Several syntax and grammar-based approaches were proposed, starting in the 1970s. In his PhD thesis, Hollander described a syntax-directed approach, involving matching patterns of IBM/360 assembly instruction sequences and converting them into the corresponding ALGOL-like instructions [19]. Schneider and Winiger described a translation-based decompiler that used the inverse of a compiler grammar intended for ALGOL-like languages [20]. Hood used a definite-clause grammar parser to decompile unoptimised Intel 8085 assembly into "SmallC", a subset of C. Inference of Pascal data types was performed using operation type, operation width and run-time bounds checks inserted by the Pascal compiler [21]. Systematic decompilation was proposed by Gorman using identifiable machine code sequences as patterns to form a grammar with which to build a VAX-to-PASCAL decompiler [22].

The 8086 C decompiling system, developed by Chen et al. from 1991 to 1993, could recover a C program from a Motorola 68000 program compiled with the Microsoft C compiler version 5. Library function recognition was performed using pre-computed unique instruction sequence matching; type inference was done via a rule-based approach [23, 24]. DisC is a C decompiler targeting x86 programs running on the DOS operating system compiled with the TurboC compiler version 2.0/2.01 [25].

Guilfanov described a type system for standard C library functions using declarations in compiler header files such as `windows.h` and a set of platform-dependent type propagation rules. The type system is implemented in the IDA Pro interactive disassembler [26]. In 2007, Guilfanov announced the launch of the Hex-Rays decompiler plugin for IDA Pro. It was mentioned that "probabilistic methods and heuristics" and data-flow analysis were in use [27].

Fokin et al. described a C++ decompiler called *SmartDec*, which is able to reconstruct polymorphic class hierarchies, types and exception handling code. For the first task, the hierarchy reconstruction is straight-forward if the run-time type information (RTTI) is available and the virtual tables (vtables) are located. Otherwise, four rules are used which look at the attributes of the vtables, virtual functions and constructors/destructors involved in order to determine the class hierarchy [28]. Basic types are treated as being composed of three components—core, i.e. integer, pointer or float, size and sign. The algorithm for inferring composite types, such as arrays and structures, operates over finite sets of labelled memory accesses [29]. The types of object pointers are treated separately, since subclass fields can be of different types for different instantiations of the same parent object. To deal with this issue, the algorithm avoids merging type information beyond the inferred object size [15]. The main task in recovering exception handling code is identifying `try` blocks and their associated `throw` and `catch` blocks. At the time of publishing, the method for reconstructing exception handling code was compiler-dependent. For `gcc`-generated code, reconstruction is done by tracking the *call sites*, or sites where an exception may be thrown, and the corresponding *landing pad*, or code that executes the `catch` block [15]. The pattern-based structural analysis used in *SmartDec* identifies and reduces constructs such as `do-while` loops, `if-then`, `if-then-else` and compound conditionals, and `break`- and `continue`-related control-flow edges [15].

2.2 Compiler model

The compiler model borrows techniques from compiler theory and program analysis, adapting it to perform decompilation.

In 1973, Housel and Halstead were the first to propose reusing compiler techniques to perform decompilation. The intermediate representation they used was called the Final

Abstract Representation, and Housel's decompiler reused concepts from compiler, graph and optimisation theory [30, 31].

Cifuentes contributed techniques for recovering function parameters and return variables using data flow analysis, and program control flow through a structuring algorithm in her 1994 PhD thesis. She demonstrated her techniques through what is considered to be the first general C decompiler for x86 programs, *dcc* [11, 32, 33]. In 1999, Mycroft was the first to consider constraint-based type inference in the context of decompilation. The use of the static single assignment form (SSA) was also suggested to "de-colour" register allocation [12]. Johnstone et al. applied well-known compiler techniques to decompile an assembly language for TMS320C6x digital signal processors, ADSP-21xx, to ANSI C. The decompilation process includes applying structural control-flow analysis to recover the call graph and control structures, and applying constant propagation and live variable analysis to recover variables [34]. Van Emmerik formally addressed the use of SSA in decompilation in his 2007 PhD thesis, describing the algorithms to convert machine code into and out of SSA, perform data flow-based type inference and resolve switch-table jump targets. These techniques were implemented in the open-source decompiler *Boomerang* [13, 8, 35]. In 2011, Lee et al. extended the work of Mycroft to consider both an upper and lower bound on variable types using type judgements. They demonstrated their method, named TIE, to be applicable as both a static and dynamic analysis [36].

Several have focussed on data-flow analysis and abstract interpretation [37] to analyse machine code. Balakrishnan and Reps demonstrated a way to perform abstract interpretation on machine code, allowing the recovery of a good approximation to the variables using the value-set domain [38, 39, 40, 5]. Their work is incorporated in a commercial tool named CodeSurfer/x86 [41]. Chang et al. proposed a modular decompilation framework so as to reuse existing source-level analysis tools, such as a model checker for C. In this framework, each decompiler is a different abstract interpreter and occupies a different level in the decompilation pipeline. Adjacent decompilers are connected via common intermediate languages and communicate via queries and *reinterpretations* [42]. Kinder proposed a simultaneous control and data flow analysis using abstract interpretation to overcome the "chicken and egg" problem in control-flow reconstruction [10]. SecondWrite is a binary rewriting tool that lifts machine code to the Low-Level Virtual Machine (LLVM) [43] intermediate representation, and makes use of the LLVM back-end to produce a C program. The lifting process involves, among other things, recovering the function prototype and deconstructing the stack into individual frames and variables. In performing these tasks, SecondWrite makes extensive use of value-set analysis (VSA) [38], for example, to extract parameters and variables. Where VSA is too imprecise, the analysis is supplemented by run-time information. Type recovery is coupled with variable recovery and the algorithm assigns a points-to set for each abstract location (aloc) as it is being discovered. Data structures are then formed by unifying the hierarchy of points-to sets [44, 45, 46].

2.3 Formal methods model

The formal methods approach focusses on two aspects: using formal methods to perform decompilation, and using decompilation as a means to verify machine code.

From 1991 to 1994, Breuer et al. demonstrated three approaches to, given the specifications for a compiler, generate a *decompiler compiler*, the inverse of compiler compilers such as `yacc`. In the first approach, the authors observe that a decompiler can be viewed as a function from object code to a list of possible source-code programs; thus decompiling equates to enumerating the attribute grammar of the compiler. A decompiler compiler for a subset of the `Occam` programming language was demonstrated [47]. In the second approach, the declarative nature of logic programming was exploited so that given a compiler specified in terms of Prolog Horn clauses, a decompiler is constructed through inverting the clauses then performing reordering to prevent non-termination [48]. The third approach defined a decompilation description language (DeCoDe), and implemented a decompiler compiler in C++ [49].

The FermaT transformation system, developed by Ward in 2000, is a C decompiler for IBM 370 assembler that uses the Wide Spectrum Language (WSL), also developed by the author, as its intermediate representation. At its core, the system performs successive transformations so that the resulting program is equivalent under a precisely defined denotational semantics. Expressions in WSL use first order logic and include, for example, existential and universal quantification over infinite sets. The system is able to derive program specifications from assembly with some manual intervention [50, 51].

Katsumata and Ohori coined the term *proof-directed decompilation* in their 2001 paper, and observed that a bytecode language can be translated to and from intuitionistic propositional logic through Curry-Howard isomorphism. They demonstrated their technique for a subset of the JVM assembly language, decompiling to typed lambda calculus [52]. In a follow-up paper, Mycroft compared type-based decompilation with proof-directed decompilation and concluded that the construction of the static single assignment (SSA) form in the former is equivalent to proof transformation in the latter for code with static jumps. It was also suggested that the two approaches may be combined by using the type-based approach as a metaframework to construct the appropriate proof transformation [53].

More recently, the work of Myreen et al. focussed on verification of ARM machine code by performing decompilation into logic. The decompilation is performed in three phases: firstly evaluating the instruction set architecture (ISA) model for each instruction stated in terms of a machine-code Hoare triple, secondly computing the control-flow graph (CFG), and thirdly composing the Hoare triples following the CFG [54, 55].

2.4 Assembly-to-C translation model

Assembly-to-C translation, which enables post-compilation optimisation and cross-platform porting of software, focusses on producing re-compilable and equivalent code, sacrificing readability.

Cifuentes et al. described the University of Queensland Binary Translator (UQBT), developed from 1999 to 2003, that enables quick porting of programs between Intel x86, Sparc and Alpha platforms. The translation involves four phases. The source-machine binary is first decoded to a source register transfer language (RTL) program using a binary-decoder; secondly, the source-RTL program is then transformed into a machine-independent higher-level RTL (HRTL) program via an instruction-decoder; thirdly, the reverse procedure transforms the HRTL program down to a target-RTL program via an instruction-encoder or alternatively into a C program; lastly, the target-machine binary is generated via a binary-encoder or an appropriate C compiler. The binary-encoder and binary-decoder are automatically generated using the New Jersey machine code toolkit [56]. UQBT performs three machine-level analyses to address three issues that face binary translation: locating all functions and all code, computing indirect control-flow targets and determining function parameters and return variables [57, 58, 59, 60].

RevNIC [61] and RevGen [62] are or incorporate assembly-to-C translators that allow fast and accurate porting of, for example, proprietary network drivers and facilitate further analysis using source-level tools. Translation is performed in two stages: firstly to the LLVM intermediate representation via either a static or dynamic binary translator, then to C via a CFG builder and the LLVM backend. The local and global state is preserved by the translator and control-flow transfers are replaced by `goto` statements.

BCR (Binary Code Reuse) is a tool that allows a malware analyst to extract and reuse, say, a decryption routine from a piece of malware. This is achieved through extracting a fragment of interest from the program and inferring its *interface*, that is, its parameters, return variables and their data dependencies. The output of BCR is a C program comprising the function's inferred prototype and its body, which consists of inlined x86 assembly. The prototype is inferred via taint information collected from multiple execution traces [63].

2.5 Information-flow model

Decompilation is viewed as using information leakage from the compilation process to reconstruct source code, and is the only one to consider an adversarial model.

Drawing inspiration from compiler optimisation, Feigin and Mycroft proposed framing compilation as a one-to-many transformation of high-security information, or the source code L, to low-security information, or the machine code M. This equivalence set of machine code programs is known as a *kernel*. The success of a decompiler relies on its ability to infer the kernel for the observed M and recover L. The larger the kernel, or the higher the degree to which the compiler (the defender) is optimising or obfuscating, the harder the decompiler (the attacker) has to work [64]. Feigin alluded to Program Expression Graphs [65], which make equivalence relations explicit in the intermediate representation, that may assist information flow-based decompilation [66]. Also noted is related work in code reverse engineering via side channel analysis [67, 68].

Viewing reverse engineering in the same vein, Della Preda and Giacobazzi proposed a theoretical framework for the obfuscation versus static analysis arms race. The observational power of the attacker, the decompiler, is characterised by the types of static analysis he/she can perform and which the defender, the compiler, must defend against [69].

This chapter has summarised prior work in decompilation in terms of five models. It is evident that recent and ongoing research encompasses all models. Popular decompilers today include Hex-Rays and Boomerang, highlighting the prominence of the heuristic-driven and compiler models, also the oldest models of the five. Introduced in the next chapter is a new model of decompilation, *decompilation as search*.

3

Decompilation as search

"When Google Translate generates a translation, it looks for patterns in hundreds of millions of documents to help decide on the best translation... The more human-translated documents that Google Translate can analyse in a specific language, the better the translation quality will be."

– Google Translate[1]

In 2004, Van Emmerik and Waddington described in some detail the process of decompiling a real-world Windows-based application. The application, which dealt with speech analysis and included heavy mathematics processing, was written and compiled with Microsoft Visual C++ and the main executable was 670KB in size. With the aid of a commercial disassembler, it took the team 415 man-hours, not including the first week of exploration, to decompile and recover 1,500 lines of source-code consisting of 40 functions and representing about 8% of the executable [70]. This effort included the development of the Boomerang decompiler [8]. The payload of the Stuxnet worm, which came to light in June 2011, took a multi-national team of numerous malware and supervisory control and data acquisition (SCADA) systems experts five months of intensive coordinated analysis to decipher [3].

This chapter takes the form of a position paper and describes the motivation and arguments for performing decompilation, which can be a tedious and time-consuming process, as search. It first examines code reuse practices in software companies, open-source software development and malware development. This is done using four methods: a survey of prior research in code reuse, a survey of software copyright litigation over the years, a study of code reuse in the popular software repository Github.com and lastly a study of code reuse in the leaked source code of the ZeuS malware. This is followed by an articulation of the proposed approach, search-based decompilation, and we will consider, as a case study, the emergence of statistical machine translation (SMT) in the area of computational linguistics and propose a research agenda for search-based decompilation. We conclude by surveying related work in search-based software engineering (SBSE), software provenance and code clone detection techniques for binary code.

[1]`http://translate.google.com/about`

3.1 How prevalent is code reuse?

Code reuse is prevalent in software companies and open-source communities, and factors such as market competitiveness and complexity of software will continue to bolster this trend.

In this first section, prior work in software reuse literature is surveyed in order to address two main questions: If software is reused, what software components are reused, and how much is reused?

3.2 Prior work in software reuse research

Code or software reuse can be defined as the use of existing software components in order to construct new systems of high quality [71] and as a means to improve programmer productivity [72]. Code reuse can be sub-divided into two categories: white-box and black-box. White-box reuse refers to the reuse of components by modification and adaptation, while black-box reuse refers to reuse of components without modification [73].

It was observed in the 1990s that software reuse, although highly desirable, does not tend to occur in a systematic fashion [71], and the success of software reuse can be attributed both to technical factors, and to a larger extent non-technical ones. Object-oriented programming (OOP) has seen the most promise for component reuse; OOP environments such as Eiffel [74] have integrated object libraries; component- and framework-based middleware technologies, such as CORBA, J2EE and .NET, have become mainstream [75].

As observed by Schmidt in 2006, the non-technical factors affecting software reuse can be summarised as belonging to the following five areas [76, 77].

Market competitiveness Schmidt observed that code reuse was more likely to be practised in business environments that were highly competitive, such as financial services or wireless networking, and where the time-to-market was crucial. On the flip side, the 1980s was an example of a period when little software reuse was achieved in the U.S. defence industry, due to the defence build-up in the Reagan administration when research funding was abundant [77]. In 2008, open-source projects were found to actively reuse software components in order to integrate functionality quickly and mitigate development costs [78].

Application-domain complexity Software components that were easy to develop, such as generic linked lists, stacks or queues, were often rewritten from scratch. In contrast, developing domain-specific components, such as real-time systems, from scratch proved too error-prone, costly, and time-consuming [76]. In a study of the IBM reusable software library (RSL), a library containing more than thirty libraries for data types, graphical interfaces, operating system services and so on, it was found that RSL's software found its way to no more than 15-20% of any software product to which RSL was an exclusive library vendor. In contrast, domain-specific components made up anywhere between 30% to 90% of the final product [79]. More recently in 2007, Mockus [80] examined the reuse of directories and files between open-source projects and analysed a sample containing 5.3 million source-code files. He found that about 50% of these files were reused at least once in different projects. The most reused files were text templates, or "PO" (Portable Object) files, used by the GNU `gettext` toolset, the install module for Perl and files belonging to internationalisation modules. The largest batch of files that were reused at least 50 times were 701 include files from the Linux kernel and 750 system-dependent configuration files from GNU libc. Haefliger [78] studied the code reuse practices amongst six open-source projects in 2008 and observed that all the projects reused external software components; a total of 55 examples were found. The reused components represented 16.9 million lines of code, whereas the number of lines of non-reused code was 6.0 million. It was discovered that 85% of reused components could be found in the Debian packages list, and the three most commonly reused components were the `zlib` compression library, the `MySQL` database and the `gtk/gdk/glib` graphical toolkit.

Corporate culture and development process A 1980s study of Japanese "software factories", which were making greater progress than U.S. firms and having greater success in reusing software components, found that the key enabler was organizational factors, such as emphasis on centralised processes, methods and tools [81]. Open source projects were found to conduct searches for suitable candidates for software to reuse via the developers' mailing list [78].

Quality of reusable component repositories In 1993, Diaz [71] argued that although the 1980s saw the creation of large-scale reuse programs, such as the U.S. Advanced Research Projects Agency's (ARPA) Software Technology for Adaptable, Reliable Systems initiative, what was missing was the lack of a standardization/certification process to maintain and ascertain software quality. Knight and Dunn [72] suggested in 1998 that a precise definition for certification of reusable software components is desirable, since a reused component can add to the quality of the new product, not just mere functionality. For open-source project developers seeking to reuse software, code quality was judged by looking at the code, reading the documentation, frequently asked questions and related articles, considering its popularity and how active its developers were [78].

Leadership and empowerment of skilled developers The quality and quantity of skilled developers and leaders determine the ability of a company or a project to succeed with software reuse. Conversely, projects that lack a critical mass of developers rarely succeed, regardless of the level of managerial and organisational support [77].

We now examine cases of GNU Public License violations and proprietary software copyright infringement in order to answer the following two questions: What software components are reused, and how is software reuse proven?

3.3 GNU Public License violations

The GPL allows certain permissions for other parties to copy, modify and redistribute software protected under the GPL as long as those parties satisfy certain conditions. In version 2 of the license, one of these conditions is that the parties provide either the "complete corresponding machine-readable source code" or a "written offer ... to give any third party ... a complete machine-readable copy of the corresponding source code" of the modified or derived work. According to Clause 4, “Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License” [82].

There are two aims in looking at GPL violations: Firstly, the types of software involved is of interest, as is the frequency of such violations in the IT industry. Secondly, cases of GPL violations are publicly documented by the Software Freedom Law Centre [83] (SFLC), the Free Software Foundation [84] (FSF), `GPL-violations.org` and third-party websites such as `Groklaw.net`.

Harald Welte, founder of `GPL-violations.org`, said that he saw 200 cases of GPL violations in Europe from 2003 to 2012 [85]. Bradley Kuhn, a technical director at the SFLC, said he found on average one violation per day for the period 21 August to 8 November 2009 [86]. Due to the large number of cases that are pending enforcement action or resolution in court, only cases where either a lawsuit was filed and won (Won) or a lawsuit was filed and settled out-of-court (Settled) were considered. To date, there has been no known case in which the terms of a GPL-styled license were not successfully upheld in court.

As of February 2013, there have been 19 documented and concluded civil lawsuits involving GPL’d software, 11 were heard in the US, 8 were heard in Europe (Table 3.1). In 16 out of 19, or 84.2% of cases, violations were found in firmware meant for embedded devices, and the GPL’d components ranged from `msdosfs`, `initrd`, the memory technology devices (MTD) subsystem, the GNU C compiler, the GNU debugger, `binutils`, `BusyBox` and packet filtering module `netfilter/iptables`.

In addition to lawsuits, there have been more than 30 examples where the terms of the GPL was successfully enforced without the need for litigation. Early cases of GPL compliance include RTLinux in 2001 [111] and Lindows in 2002 [112]. In 2004, Welte was said to have coaxed several distributors, including Fujitsu-Siemens, Securepoint, Belkin and Asus, into compliance [113]. In 2005, a further 13 companies, including Motorola, Acer, AOpen, Micronet, Buffalo and TrendWare, were handed warning letters by Welte [114]. Welte’s `gpl-violations.org` project managed to conclude more than 25 amicable agreements, two preliminary injunctions and one court order for the period January to March 2005 alone [114].

It was not always clear how the GPL’d software components were discovered. In the Viasat/Samsung case a Linux developer discovered code belonging to BusyBox, parts of which he had written [115]. In the *Mistic software v. ScummVM* case, the ScummVM developers were able to reproduce a same 0.9.0-related `gfx` glitch present in one of the games [116]. In one case, *Welte v. Fortinet Ltd.*, encryption was used, possibly to hide the presence of GPL’d code.

3.4 Proprietary software copyright infringement

There are seven known cases involving proprietary software incorporating other proprietary software. In the case of *Cadence Design Sys. v. Avant!* the nature of the software in question is not documented; however what is known is that both companies were chip-design software manufacturers [117]. Three of the cases dealt with operating systems and programming-related software: *Green Hills v. Microtec* dealt with compilers [118], the

Year	Case	Product	GPL component	Outcome
2002	*Progress Software v. MySQL*	NuSphere Gemini Table	MySQL	Settled [87]
2004	*Welte v. Allnet*	Router firmware	netfilter/iptables	Settled [88]
2004	*Welte v. Gigabyte Technology*	Router firmware	Linux, netfilter/iptables	Settled [89]
2004	*Welte v. Sitecom*	Router firmware	netfilter/iptables	Won [90, 91]
2005	*Welte v. Fortinet*	FortiOS, Firewall, Antivirus	Linux kernel	Won [92]
2006	*Welte v. DLink*	Data storage device firmware	Linux kernel (msdosfs, initrd, mtd)	Won [93, 94]
2007	*SFLC v. Monsoon Multimedia*	HAVA Firmware	Busybox	Settled [95]
2008	*SFLC v. High-Gain Antennas*	Firmware	Busybox	Settled [96]
2007	*SFLC v. Xterasys*	Firmware	Busybox	Settled [97]
2008	*SFLC v. Verizon*	FiOS firmware	Busybox	Settled [98]
2008	*SFLC v. Extreme Networks*	Firmware	Busybox	Settled [99]
2008	*SFLC v. Bell Microproducts*	Firmware	Busybox	Settled [100, 101]
2008	*SFLC v. Super Micro Computer*	Firmware	Busybox	Settled [102, 99]
2008	*Welte v. Skype Technologies*	VOIP Phone firmware	Linux kernel	Won [103]
2008	*FSF v. Cisco*	Router firmware	GCC, binutils, GDB	Settled [104, 105]
2008	*Jin v. IChessU*	IChessU client	Jin	Settled [106]
2009	*SFLC v. JVC, Samsung, Best Buy et al.*	Cell phones, PDAs and other small, specialized electronic devices	Busybox	Settled/Won [107, 108]
2009	*EDU 4 v. AFPA*	EOF	VNC	GPL infringement successfully appealed by AFPA [109]
2011	*AVM v. Cybits*	Router firmware	Linux kernel, iptables	GPL parts ruled modifiable [110]

Table 3.1: Known cases of GPL violations either settled or won.

Cisco v. Huawei case involved embedded systems [119, 120], while *Compuware v. IBM* dealt with mainframe systems [121, 122], and *Veritas v. Microsoft* dealt with the disk manager in the Windows 2000 operating system [123, 124]. Database management software was also featured in two of the cases, *Computer Associates (CA) v. Quest Software* [125] and *CA v. Rocket Software* [126, 127] (Table 3.2).

In *Veritas v. Microsoft* and *Compuware v. IBM*, the source code was shared under a non-disclosure agreement. In the similar case, *Green Hills v. Microtec*, source code was shared under a distribution agreement. In *CA v. Quest* the source code was allegedly obtained by former CA employees [125]. In *Cisco v. Huawei* the evidence presented was that of the similarity of the user interface commands, but crucially also that of a similar software bug that existed in both the versions of IOS and VRP in use at the time.

Year	Case	Infringing product	Infringed product	Outcome
1993	*Green Hills v. Microtec*	"compiler technology"	"compiler technology"	Settled [118]
1995	*Cadence Design Sys. v. Avant! Corp.*	*Unknown*	*Unknown*	Settled [117]
1999	*McRoberts Software v. Media 100*	Comet/CG (character generation) source code	Finish product line	Won [128]
2002	*Compuware v. IBM*	File Manager, Fault Analyser	File-AID, Abend-AID, Xpediter	Settled [121, 122]
2003	*Cisco v. Huawei*	Quidway Versatile Routing Platform (VRP)	Internetwork Operating System (IOS)	Settled [119, 120]
2005	*CA v. Quest Software*	*Unknown*	Quest Central for DB2	Settled [125]
2006	*Veritas v. Microsoft*	Logical Disk Manager	Volume Manager	Settled [123, 124]
2009	*CA v. Rocket Software*	"DB2 products"	"DB2 products"	Settled [126, 127]

Table 3.2: U.S. cases involving proprietary software incorporating proprietary software. In cases where the product in question is not known, it is labelled as *Unknown*.

3.5 A study of code reuse on Github

Github.com is the leading repository for open-source code bases, hosting a large variety of projects in dozens of languages, including code belonging to several well-known software companies, such as Mozilla, Twitter, Facebook and NetFlix. As an open-source repository, Github had 1,153,059 commits from January to May 2011, more commits than Sourceforge, Google Code and Microsoft's CodePlex combined [129]. The purpose of studying code reuse on Github was to determine the components most widely reused. Two methods were used to study the two forms of code reuse: to study white-box reuse, the most popularly forked projects were examined and a keywords-based search on Github and Google was performed; to study black-box reuse, a keywords-based search was performed.

This study found that projects centred on web servers and related tools saw more than 2,000 project instances of white-box code reuse. Operating system and database development were also areas where white-box reuse was actively pursued. The level of black-box reuse was significantly higher than that of white-box reuse. Black-box code reuse in Github was centred around: installation and programming tools (21,000 instances), web servers and related tools (2,000 instances), databases (1,400 instances), cryptographic libraries (885 instances) and compression libraries (348 instances). Some specialised libraries that were commonly reused were the Boost C++ library for C++ projects, `libusb` for embedded devices, and `libsdl` for game development.

White-box reuse Forking a project on Github means to make a copy of that project to modify it. The number of forks that a project has is a good proxy not only for the popularity of that project, but also for white-box code reuse. The most-forked repository as of February 2013 was Twitter's `Bootstrap` front-end web development framework with 12,631 forks. Of the top ten most popularly forked repositories, seven of them were for web-based development. The three non-web-based code bases were a demonstration-only repository, a customised configuration manager for the `zsh` shell program and a development toolkit for mobile platform applications (PhoneGap). The most popular C/C++-based repository on Github was the Linux kernel with 1,933 forks (Table 3.3).

Rank	Repository	Description	Forks
1	Bootstrap	Front-end web framework	12,631
2	Spoon-Knife	Demonstration repository	12,249
3	homebrew	A package manager for Mac OS	5,477
4	rails	A Ruby-based web framework	5,014
5	html5-boilerplate	An html5-based web framework	3,821
6	hw3_rottenpotatoes	A movie rating website	3,598
7	oh-my-zsh	A configuration management framework for the zsh shell	3,310
8	node.js	A Javascript-based event-based I/O library	3,305
9	jQuery	A Javascript-based library	3,217
10	phonegap-start	A tool for PhoneGap application development	2,720
11	impress.js	A Javascript-based presentation library	2,442
12	hw4_rottenpotatoes	A movie rating website	2,343
13	backbone	A Javascript-based database library	2,284
14	d3	A Javascript-based visualisation library	2,103
15	jQuery-ui	A user-interface library for jQuery	2,044
16	symfony	A PHP-based web framework	2,043
17	game-of-life	A demonstration application for the Jenkins build management tool	2,008
18	Linux	A UNIX-based kernel	1,960
19	CodeIgniter	A PHP-based web framework	1,951
20	phonegap-plugins	Plugin development for PhoneGap	1,900

Table 3.3: The most forked repositories on Github (as of February 2013).

A second independent indicator for white-box reuse was the number of search results for the phrase "based on". This search returned 269,789 out of 5,379,772 repositories and 28,754,112 hits in all indexed source-code on Github as of February 2013. Since the occurrence of this phrase on its own does not imply code reuse, for example "based on position information" is a false positive, these search results were further analysed by including additional terms, such as well-known project names and platforms, into the query. A similar query was made on the Google search engine using the site-specific qualifier `site:github.com` and the gross and unique number of hits were noted. The results, tabulated in Table 3.4, are sorted by the number of Github repositories found. Searches that returned fewer than 1,000 hits were excluded.

It was observed that web-based projects occupied the top-ranked spots in both of these tables, in particular the `jQuery`, `Bootstrap`, `Node.js` and `Rails` projects. This can be explained by the fact that the most popular programming language on Github is Javascript, with 21% of all projects written in the language. The other popular languages were Ruby (13%), Java (8%) and Python (8%). One possible implication of this is that white-box reuse is most popular amongst web-based software developers. The second category of software projects in which white-box reuse occurred involved operating systems projects, especially for Linux-related projects. The high number of hits for Linux can also be attributed to the fact that many repositories had a directory named "Linux", thus each occurrence of a file within a "Linux" directory produced a hit in the search results.

The relevant results for the phrase "based on Google" were "Google Web Toolkit", "Google File System", "Google Closure Library", and "the Google Chromium project". The top entry for the Linux-related queries was "based on the Linux kernel" with 2,710 hits. One phrase that had 1,790 occurrences was "Based on `linux/fs/binfmt_script.c`" found in the file `linux/fs/binfmt_em86.c`. The Ubuntu-related entries were related to configuration files and setup scripts, for example “based on the Ubuntu EC2 Quick Start Script”, “install_dependencies.sh (based on the Ubuntu script)”, “based on the Ubuntu /etc/init.d/skeleton template”, and “based on Ubuntu apache2.conf”. MySQL-related results were databases, parsers, lexers and scripts based on the MySQL code base. Database projects associated with MySQL included “MariaDB”, WikiDAT”, “NodeDB”, “MySQL-for-Python3”, “Twitter MySQL”, “GoogleCloudSQL”, “Shinobi” and “Smart Library Management System (SLMS)”, just to name a few. There was only one relevant Windows-based result—“based on Windows winerror.h”. For queries related to GNU, “based on the GNU Multi-Precision library” was the most commonly occurring phrase, followed by “based on the GNU toolchain”. Popular Mozilla projects and software that were of interest to developers were “Mozilla XUL”, “Mozilla Rhino Javascript engine”, “Mozilla Jetpack” and a “Mozilla SHA1 implementation”.

False positives On manual inspection of the other results, it was often the case that the phrase “based on X” referred to X as a platform, an API or an SDK, rather than an instance of code reuse. Examples of this were statements such as “the server is based on Windows” and “the development is based on Windows, therefore…”. A fair proportion were statements alluding to the use of a certain API or SDK, such as “based on the Google App Engine SDK” and “based on Windows IMM32 API”. Android-related hits were primarily associated with Android application development, and included projects dealing with “Android NDK”, “Android Maven Plugin”, “Android SDK” and “Android example project from Cordova/PhoneGap”. Windows-related entries were mostly associated with APIs, SDKs and demonstration code. A commonly occurring phrase for the Apple-related query was “(Based on Apple Inc. build …)”. This phrase was printed to standard output whenever the command `llvm-gcc --version` was executed, and hence it was found in numerous issues filed on Github.

Black-box reuse The next set of experiments were performed to investigate black-box reuse on Github. To do this, a list of libraries from the Debian distribution with the most reverse dependencies was obtained by running the command `apt-cache dotty` on an Ubuntu 11.10 system. If a package A is dependent on B, then we say that B is reverse dependent on A. Out of a total of 35,945 libraries, the 10 with the most reverse dependencies are listed in Table 3.5.

The number of repositories having the phrase “apt-get install” was used as an indication of the number of projects that listed dependency information. There were 11,033 repositories according to Github search.

Table 3.6 lists the results searching for “apt-get install ¡library¿” on Github. The top entries were associated with installation of build tools (`make`, `g++`, `build-essential`), language-specific packages (`ruby`, `python-dev` and `php`). Tools such as `wget` and `git` were also commonly used to download additional packages not maintained by the distribution. The popularity of Ruby- and Python-based projects explains the high occurance of `ruby`, `rubygems`, `python-dev` and `python-support` packages. The popularity of web-based applications on Github was evident given the number of repositories using `apache2`, `libxml2`, `libxslt`, `libssl` and `nginx`. Databases were also commonly listed as dependencies, for example `mysql` and `postgresql`. The `zlib` compression library was also included as a requirement in 348 repositories.

Search phrase	Github (repositories)	Google (unique entries)
“based on (the) jQuery”	8,336 (683)	293,400 (650)
“based on (the—Twitter’s) Bootstrap”	3,005 (623)	115,117 (460)
“based on (the) Rails”	11,441 (479)	699,700 (320)
“based on (the) Google”	3,783 (426)	345,700 (630)
“based on (the) Node”	38,527 (367)	2,346,700 (590)
“based on (the) Linux”	291,490 (334)	2,318,600 (220)
“based on (the) Ubuntu”	2,080 (286)	44,800 (270)
“based on (the) Android”	4,292 (259)	106,230 (260)
“based on (the) MySQL”	1,809 (249)	351,700 (140)
“based on (the) Windows”	12,568 (227)	604,650 (150)
“based on (the) CodeIgniter”	3,400 (195)	3,350 (260)
“based on (the) Symfony”	3,345 (154)	28,430 (200)
“based on (the) Debian”	2,749 (145)	121,440 (220)
“based on (the) GNU”	23,825 (129)	11,010,600 (150)
“based on (the) Gtk”	10,498 (90)	1,920 (120)
“based on (the) BackBone”	1,797 (88)	175,660 (230)
“based on (the) Apple”	10,275 (70)	240,800 (610)
“based on (the) Microsoft”	6,314 (49)	194,430 (200)
“based on (the) Gnome”	3,706 (45)	178,620 (120)
“based on (the) Mozilla”	16,595 (42)	571,250 (350)
“based on (the) BSD”	15,810 (37)	369,890 (120)
“based on (the) zlib”	24,973 (16)	54,620 (110)
“based on (the) FreeBSD”	6,241 (14)	21,170 (130)
“based on (the) Berkeley”	1,386 (6)	69,800 (110)

Table 3.4: Number of results returned for Github search and Google search (using site:github.com) sorted by the number of Github repositories.

	Package	Description	Reverse dependencies
1	libc6	GNU C library	14,721
2	libstdc++6	GNU standard C++ library	4,089
3	python	An interactive high-level object-oriented language	4,009
4	libgcc1	GNU C compiler support library	3,811
5	libglib2.0	GLib library of C routines	2,758
6	libx11-6	X11 client-side library	1,582
7	libgtk2.0	GTK+ graphical user interface library	1,464
8	libqtcore4	Qt 4 core module	1,221
9	zlib1g	Compression library (runtime)	1,205
10	python-support	Automated rebuilding support for Python modules	1,017

Table 3.5: Libraries with the most reverse dependencies in Ubuntu 11.10. If a package A is dependent on B, B is reverse dependent on A.

The `libboost` library was popular among C++-based projects. Given that the `g++` compiler and `libboost` were mentioned about 2,000 and 260 times respectively, it is estimated that `libboost` was used in about 13% of C++ projects. The term “boost” was found in 2,816 C++ repositories, which is about 14% of C++ repositories. Another popular C++ library was `libqt4`, which is a cross-platform application framework.

More specialised libraries in the list were `libusb` and `libsdl`. The `libusb` library provides an interface that gives applications access to USB-enabled embedded devices. Popular devices included the Microsoft Kinect, Android devices and Apple iPhones. The `libsdl`

Package	Github repositories	Package	Github repositories
make	5,620	dpkg	329
git	4,506	automake	266
g++	1,994	libxslt	265
ruby	1,785	libboost	263
python-dev	1,414	libsqlite3	261
build-essential	1,158	subversion	261
curl	1,043	libtool	260
wget	1,036	bison	223
python-support	970	python2.7	194
gcc	966	ant	179
tar	939	libqt4	173
mysql	924	flex	169
php	810	aptitude	163
libxml2	564	libc6	139
openssl	504	python2.6	139
apache2	501	rsync	139
rubygems	486	libusb	138
postgresql	442	locales	133
libssl	384	libsdl	133
nginx	381	openjdk-6-jdk	126
zlib	348	adduser	122
autoconf	330	libpng	122

Table 3.6: Number of occurances of common library names returned by Github search using "apt-get install ¡library¿".

library is a multimedia-focussed library that exposes an interface to graphics, sound and input devices for game development.

3.6 Code reuse in malicious software

Analysis of the ZeuS source code suggests that code reuse is prevalent in malware development.

Studying code reuse in malware development is interesting because, at first surprisingly, companies producing malware operate like standard commercial software firms. One such company, Innovative Marketing Ukraine, was reported to have reaped US$180 million in profits in 2008 [130].

The ZeuS trojan is a well-known malware family specialising in stealing on-line banking credentials and automating parts of the money-laundering process. ZeuS is not used by its developers but is sold as a toolkit to cybercrime rings doing the actual money laundering. Groups working on toolkits like ZeuS adopt a normal software development lifecycle and employ QA staff, project managers and developers for specialised components such as kernel drivers, sophisticated packers and graphical user interfaces [131]. In May of 2011, a version of the source code for the ZeuS toolkit, worth as much as US$10,000 per copy, was leaked [132, 133]. A copy of the source code is available on Github[2]. Two aspects of the source code were studied, namely, the amount of actively developed software within

[2] `https://github.com/Visgean/Zeus`

the ZeuS source code and the extent to which software was obtained from external sources. The amount of actively developed software components was studied by looking at the "last modified" dates in the file metadata; a list of externally obtained software was obtained via a search for the string "`http://`" in the source code.

Analysis of the ZeuS source code suggests that 6.3% of the files, or 5.7% of the code, was worked on post-October 2010. If work did indeed begin in October 2010, then 94.3% of version 2.0.8.9 was "legacy" code. The 5.7% that was new involved functionality such as data interception, encryption and storage in the *client* component. A study of the web references embedded in the comments throughout the entire source code tree revealed a total of 3,426 lines (3.7%) of white-box code reuse throughout the whole code base, and 8,180 out of 140,800 bytes (5.8%) of black-box code reuse considering only the 32-bit *client* executable. However, this is a conservative estimate. Categories of software components reused in ZeuS 2.0.8.9 included: keyboard symbols-to-unicode encoding (xterm), VNC (UltraVNC), random number generation (Mersenne Twister), data compression (info-zip, UCL) and binary analysis (BEAEngine).

The leaked ZeuS source code was version 2.0.8.9 of the toolkit, and is made up of 68,975 lines of C++ and 22,914 lines of PHP. The toolkit is divided into six components: *bcserver*, *builder*, *buildtools*, *client*, *common* and *server*. Comments are in Russian encoded using the Windows-1251 Cyrillic code page. The *client* has three main functions: obtaining login credentials, transferring the credentials to the botmaster's server and ensuring its continued presence on the host. The *bcserver* component stands for the "back connect server", and it also exists as a Microsoft Visual Studio 2003 C++ project. The *server* component, which is written in PHP, is responsible for storing stolen credentials in a MySQL database.

3.6.1 Analysis of "last modified" dates

The oldest and newest "last modified" dates were "14 October 2010" and "10 May 2011" respectively. Out of the 126 C++ and PHP files in the source code, 118 or 93.7% had the same oldest last-modified date. In terms of code, these files made up 94.3% of the code base. It is possible that these files were unmodified from the previous release when the timestamps were last updated. The breakdown of these files was as follows.

> *bcserver* (3 out of 3 or 100%),
>
> *builder* (7 out of 9 or 77.8%),
>
> *buildtools* (4 out of 4 or 100%),
>
> *client* (28 out of 32 or 87.5%),
>
> *common* (32 out of 34 or 94.1%) and
>
> *server* (44 out of 44 or 100%).

The files that had modified-by dates after October 2010 were:

> `source/builder/buildconfig.cpp` (sets parameters such as the encryption key and *server* URL in the *client*, last modified 4 November 2010),

`source/builder/info.cpp` (information dialog box in *builder*, last modified 2 April 2011),

`source/client/core.cpp` (core functionality of the *client*, last modified 29 October 2010),

`source/client/cryptedstrings.cpp` (list of hard-coded obfuscated strings, last modified 14 April 2011),

`source/client/httpgrabber.cpp` (HTTP/HTTPS data interception, last modified 5 November 2010),

`source/client/wininethook.cpp` (Hooking of `wininet.dll`, last modified 3 November 2010),

`source/common/binstorage.cpp` (Encrypted file management on the *client*, last modified 5 November 2010) and

`source/common/crypt.cpp` (Functions dealing with cryptography and random number generation, last modified 5 November 2010).

Most of the development on the *client* appears to have been from October to November 2010; the *builder* component was worked on up until 2 April 2011.

A 30-line function in `client/core.cpp`, `_getKernel32Handle`, was found to contain comments in English (Figure 3.1). This was unusual given that most of the comments were in Russian. A search on Google for the comment string "clear the direction flag for the loop" revealed a post on a software security-related blog dated 19 June 2009 entitled "Retrieving Kernel32's Base Address" [134]. The blog post described three methods of retrieving the base address of the `kernel32.dll` image in memory. The code snippet which was copied was the third of these methods and this method was, according to the blog post, "more robust" in dealing with Windows 2000 up to Windows 7 RC1.

Analysis of the files modified post-October 2010 seemed to suggest that the changes focussed on the *client*, and concentrated in the areas of data interception, encryption and storage, possibly to deal with newer versions of Windows. The files belonging to the *server* were untouched.

3.6.2 Search for "http://" string

The second method used to study code reuse was to search for the "http://" string in the source code as an indicator of publicly available information used by the developers of ZeuS. The results are listed in Table 3.7.

Web references were found in the *client* (1–4), *server* (5–6) and *common* components (7–11), and can be classified into four categories—network connectivity and communications (1–4,6,10–11), data compression (5,9), binary analysis (7) and random number generation (8).

SOCKS The two SOCKS-related URLs found in the source code referred to the SOCKS RFC specification, not to actual source code. A quick search for the file name `socks5-server.cpp` and a variable name `S5_CLIENT_IS_IPV6` yielded no results on Google.

```
HMODULE _getKernel32Handle(void)
{
#if defined _WIN64
  return NULL; //FIXME
#else
  __asm
  {
    cld //clear the direction flag for the loop

    mov edx, fs:[0x30] //get a pointer to the PEB
    mov edx, [edx + 0x0C] //get PEB->Ldr
    mov edx, [edx + 0x14] //get the first module from the InMemoryOrder module list

  next_mod:
    mov esi, [edx + 0x28] //get pointer to modules name (unicode string)
    mov ecx, 24 //the length we want to check
    xor edi, edi //clear edi which will store the hash of the module name

  loop_modname:
    xor eax, eax //clear eax
    lodsb //read in the next byte of the name
    cmp al, 'a' //some versions of Windows use lower case module names
    jl not_lowercase
    sub al, 0x20 //if so normalise to uppercase

  not_lowercase:
    ror edi, 13 //rotate right our hash value
    add edi, eax //add the next byte of the name to the hash
    loop loop_modname //loop until we have read enough

    cmp edi, 0x6A4ABC5B //compare the hash with that of KERNEL32.DLL
    mov eax, [edx + 0x10] //get this modules base address
    mov edx, [edx] //get the next module
    jne next_mod //if it doesn't match, process the next module
  };
#endif
}
```

Figure 3.1: The code listing for function `getKernel32Handle`, which was posted on a blog in June 2009 [134].

xterm The `xterm` reference was an indication that the source code of the terminal emulator for the X Windows system was used in the VNC server implementation in the *client*. On inspection of the source code of `xterm`, it was discovered that the `keysymtab[]` array from `key2symucs.c`, which occupied 800 lines of C, was reproduced verbatim in `client/vnc/vnckeyboard.cpp`. The `keySymToUnciode` (sic) function in the same file, comprising 45 lines of C, was a close derivative of the `keysym2ucs` function. Attribution to the original author (Markus G. Kuhn ¡mkuhn@acm.org¿, University of Cambridge, April 2001) had been removed, presumably to obscure the origins of the code.

UltraVNC Based on the reference to `UltraVNC` in the ZeuS VNC module, the source tar archives of `UltraVNC` versions 1.0.2, 1.0.8.2 and 1.1.8.0 were analysed for signs of reuse. Although the header file in which the reference was found, `rfb.h`, bore resemblance to

	URL	Description
1	`http://www.opennet.ru/base/net/socks5_rfc1928.txt.html`	Specification for the SOCKS5 protocol (*client*)
2	`http://www.sockschain.com/doc/socks4_protocol.htm`	Specification for the SOCKS4 protocol (*client*)
3	`http://invisible-island.net/xterm`	The xterm emulator used by the VNC server (*client*)
4	`http://www.uvnc.com`	UltraVNC (*client*)
5	`http://www.info-zip.org/Zip.html`	A data compression program used to archive files (*server*)
6	`http://etherx.jabber.org/streams`	Part of the Jabber API to interact with the Jabber Instant Messenging network (*server*)
7	`http://beatrix2004.free.fr/BeaEngine`	An x86 disassembler (*common*)
8	`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`	Mersenne Twister random number generator (*common*)
9	`http://www.oberhumer.com/opensource/ucl`	A data compression library (*common*)
10	`http://www.google.com/webhp`	URL used to test network latency (*common*)
11	`http://www.winehq.org/pipermail/wine-bugs/2008-January/088451.html`	A description of a bug in the way WINE handles the WriteConsole Windows API call (*common*)

Table 3.7: Results of a search for "http://" in the source code of ZeuS version 2.0.8.9.

the `rfb/rfbproto.h` file in the source tree, there was no evidence of verbatim copying of `UltraVNC` code. If it is assumed that ZeuS' RFB implementation was based on the UltraVNC version, the number of lines reused was 1,216.

info-zip With regards to the reference to `http://www.info-zip.org/Zip.html`, on inspection of the PHP script in question, `server/php/fsarc.php` (server file system archival), the following code was found.

```
$cli = 'zip␣-r␣-9␣-q␣-S␣"'.$archive.'"␣"'.implode('"␣"', $files).'"';
exec($cli, $e, $r);
```

A copy of `zip.exe`, with an MD5 hash of `83af340778e7c353b9a2d2a788c3a13a`, corresponding to version 2.3.2,was found in same directory.

Jabber To investigate the implementation of the client for the Jabber instant messaging protocol, two searches were made: firstly for the file name in question, `jabberclass.php`, then for unique strings in the script. Both searches came up with nothing pointing to earlier, public versions of the script. Thus, it was concluded that the Jabber client was implemented from scratch.

BEAEngine The reference to the `BEAEngine` was found in `common/disasm.h`, which contained three functions: `Disasm::init`, `Disasm::uninit`, and `Disasm::_getOpcodeLength`. An external call to an `LDE` function was found in the last of these functions. The external call to `LDE`, or Length Disassembler Engine (LDE), is statically linked to the object file `lib/x32/lde32.lib`.

Mersenne Twister A URL was found in `common/crypt.cpp` which made reference to the Mersenne Twister pseudo-random number generator developed by Makoto Matsumoto and Takuji Nishimura. The C source code of the Mersenne Twister was available from the website. Comparison of the two sets of code revealed that 62 lines of C was copied verbatim from the original source code, namely the `init_genrand` and `genrand_int32` functions.

UCL The next file that was investigated was `common/ucl.h`. Both the URL for the UCL data compression library and the string "1.03" were found. Manual analysis of both implementations suggests that the ZeuS was a syntactically-modified version of the original. Examples of syntactic changes included: replacement of the `ucl_uint` and `ucl_bytep` types with `DWORD` and `LPBYTE` respectively, replacement of calls to `ucl_alloc` and `ucl_free` with `Mem::alloc` and `Mem::free` respectively, removal of all calls to `assert` and removal of all statements defined by the `UCL_DEBUG` preprocessor directive variable. The ZeuS-variant of the UCL library included a total of 1,303 lines of C. This was an example of large-scale white-box reuse in ZeuS.

Google/webhp, WINE The last two references were not directly related to code reuse. Requests to `www.google.com/webhp` were used to test the latency of the network connection as part of the *client* report back to the *server*. The reference to the WINE bug suggests that at least one developer was using a Linux machine to test the *buildtools* or *bcserver*.

3.7 Summary of findings

The findings of the four surveys may be summarised as the following.

> Software reuse is widely practised in open-source projects, commercial projects and malware development alike, with black-box reuse favoured over white-box reuse. The exception is web-related projects, which are more likely to perform white-box than black-box reuse.
>
> Reused software can make up between 30–90% of the final product.
>
> Widely reused software include operating systems software, such as the Linux kernel, software development tools and libraries, such as the GNU C library, database software, encryption libraries and compression libraries.
>
> This trend is likely to continue, bolstered by factors such as market competitiveness, the growing complexity of software products and the availability of high quality, public source code.

3.8 Proposed approach: Search-based decompilation

Building upon the observation that code reuse is prevalent, this thesis proposes framing decompilation as a search problem. It is noted that a similar shift occurred in machine translation from a rule-based approach to statistical machine translation (SMT). What follows is an examination of the factors leading to the emergence of SMT, its benefits as well as its challenges and a research agenda for search-based decompilation is proposed.

3.8.1 Case study: Statistical machine translation

Statistical machine translation provides valuable insight into the benefits as well as the challenges that search-based decompilation can learn from.

In April of 2006, Google switched its Translate system from Systran to its own machine translation system. Google's Franz Och explained the switch.

> "Most state-of-the-art commercial machine translation systems in use today have been developed using a rule-based approach and require a lot of work by linguists to define vocabularies and grammars. Several research systems, including ours, take a different approach: we feed the computer with billions of words of text, both monolingual text in the target language, and aligned text consisting of examples of human translations between the languages. We then apply statistical learning techniques to build a translation model." [135]

Although the idea of applying statistics and cryptanalysis to language translation was first proposed by Weaver in 1949 [136], it was only in 1990 that Brown et al. [137] described the main ideas for statistical machine translation (SMT) in what is now widely considered to be the seminal work of the field. The re-emergence of SMT could be attributed to several factors, including: the growth of the Internet, which caused a spike in demand for the dissemination as well as the assimilation of information in multiple languages, the availability of fast and cheap computing hardware, the development of automatic translation metrics, the availability of several free SMT toolkits and the availability of a corpus of parallel texts [138].

A similar situation exists for a proposed search-based decompilation (SBD) approach: there is a pletora of open-source projects such as GNU, the Apache foundation, Linux, BSD distributions, as well as public code repositories such as Google Code, Github, Sourceforge; the corpus of parallel "texts" can be obtained through compilation using the most widely used `gcc` and `g++` GNU compilers. What is needed, therefore, is an SBD methodology, and a quality metric for the decompiled output. As demonstrated in the *Cisco v. Huawei* case, software bugs form a fingerprint which can uniquely identify a software component and thus provide legal evidence in court. Such fingerprinting could prove useful also as a quality metric for SBD and is thus an interesting line of research to pursue.

The goal of SMT is to take a sentence in the source language and transform it into a linguistically equivalent sentence in the target language using a statistical model. Broadly speaking, this model is composed of three components: a language model, which given an English string e assigns $P(e)$, the probability of occurrence of e in the corpus; a translation model, which given a pair of strings f, e assigns $P(f \quad e)$, the conditional probability that f occurs given e; and a decoding algorithm, which given a language model, translation model and a new sentence f, finds a translation e that maximises $P(e) \quad P(f \quad e)$. The translation model may be word-based or phrase-based with the word-to-word or phrase-to-phrase correspondences modelled using an alignment procedure. Phrase-based SMT is considered to be superior to word-based SMT [139].

One benefit of SMT is its non-reliance on linguistic content. Unlike rule-based translation, SMT systems are not tailored to any specific pair of languages, and developing a new SMT system can be as short as a day [140]. Oard and Och reported constructing a Cebuano-to-English SMT system in ten days [141].

Secondly, SMT systems tend to produce natural translations instead of literal translations. For example given "Schaukelstuhl", the intuition is that "rocking chair" has a higher $P(e) \quad P(f \quad e)$ than "rockingstool".

A fundamental research question for SBD is "What constitutes a token?", since binary code is different and possibly more complex, as it can include, for example, conditional control flow and recursion. For text, tokens are typically words or phrases; for binary code, potential candidates are instructions, instruction sequences or even graphs.

One challenge faced by phrase-based SMT is that statistical anomalies may occur, e.g. "train to Berlin" may be translated as "train to Paris" if there are more occurrences of the latter phrase. Out-of-vocabulary words are words that do not occur in the training set, such as names, and hence cannot be translated. Koehn suggested that nouns be treated separately with more advanced features and modelling [142].

Another challenge for phrase-based SMT is the inability to produce correct syntax at the sentence level whilst having correct local structure, sometimes resulting in incoherent sentences. This is due in part to the inability of phrase-based SMT to handle large-scale reordering. One proposed solution is to perform syntax transformation prior to phrase-based SMT [142].

A third challenge for SMT is its dependence on the corpus and results can be poor for a corpus that is sparse, unrepresentative or of poor quality [143]. This problem can only be addressed by including more training data.

The corresponding potential challenges for SBD are in dealing with possible "out-of-vocabulary" instructions or instruction sequences, the ability to perform large-scale re-ordering of decompiled source-code, and its dependence on the quality of the "corpus", which can vary with the platform and compiler used. It is perhaps still premature to consider these issues at the onset without an initial SBD methodology. Given that external software components can make up as much as 90% of the final product, it is practical to initially focus on whole-function matching. Moreover, function-level decompilation is the default use-case of both the Boomerang [8] and Hex-Rays [7] decompilers.

3.8.2 Proposed research agenda

To summarise, the following research agenda for SBD is proposed.

Drawing inspiration from SMT, one possible starting point is to investigate a token-based approach to SBD, and address the question "What constitutes a token?", in the context of binary code.

It is natural to focus on whole-function matching as a starting point. Depending on its success, other forms of matching, such as basic block matching or call-graph matching, may be subsequently considered.

Based on the observation that software bugs form a fingerprint that uniquely identifies software, investigating software testing as a means to fingerprint and evaluate the quality of the decompiled output is a viable avenue of research.

The GNU C library and the Linux kernel are useful as an initial SBD corpus, since it is widely used by open source and commercial projects alike.

3.9 Related work

Search-based software engineering (SBSE). SBSE considers the forward problem of using search techniques, such as operations research and evolutionary computation, to generate programs, often in a high-level language, that meet one or more success criteria [144]. Harman et al. provide two comprehensive surveys of the field [145, 144]. SBSE considers two main components in its methodology: a suitable abstraction for the problem that lends itself to search, and a fitness function through which solutions can be evaluated [145]. The applications of SBSE are wide-ranging and include testing and debugging, maintenance, software design and requirements and specification engineering [144]. The SBSE community has thus far not considered decompilation as one of its focus areas. It is noted, however, that the abstraction-fitness framework of SBSE is also relevant for SBD.

Binary code provenance. Davies et al. coined the term “Software Bertillonage” to refer to the problem of determining the origin of software components, in their case Java class archives. Analogous to human fingerprints and mugshots, their index, called *anchored class signatures*, comprised the type signatures of the class, its default constructor and its methods [146]. Rosenblum et al. studied software authorship attribution using instruction- and control flow-based representations of binary code. They were able to correctly identify the author out of a set of 20 with 77% accuracy, and rank the correct author among the top five 94% of the time, showing that programmer style is preserved through the compilation process [147]. Using similar techniques, Rosenblum et al. were also able to jointly infer source language, compiler family and version, and optimisation level options used to produce a binary with 90% accuracy [148]. While the approaches to provenance are relevant to SBD, the goals are different in that the focus of SBD is on matching functionality, rather than artefacts. Unlike the work of Davies et al., focussing on machine code, rather than on Java bytecode, is favoured.

Binary code clone detection. Unlike determining provenance, which aims to track software across multiple applications, the aim of binary code clone detection is to track duplicated code within the same application, or between closely related versions of the same application. The Binary Matching tool (BMAT), developed by Wang et al., was able to successfully match different versions of the same dynamically linked library (DLL) using branch prediction and code coverage information [149]. Dullien and Rolles extracted vulnerability information by performing a “diff” on the patched and unpatched versions of the same executable. Their technique relied on iteratively determining pairs of fixpoints in the call graph and control-flow graph. Basic blocks were matched using the method of *small primes product* to overcome the problem of instruction reordering [150]. Sæbjørnsen et al. developed a technique that performed fast matching of binary code clones using an instruction-based representation [151]. Hemel et al. used string literals to identify binary code clones. The advantage of this method was that, unlike instruction- and control flow-based techniques, disassembly was not required [152]. Like software provenance, the approaches here are also relevant to SBD, but the goal is to match the same function across different applications. A key issue that SBD faces is compiler-related code transformations, which binary clone detection does not traditionally deal with.

As a starting point for investigating search-based decompilation, the next chapter addresses the question ”What constitutes a token?”, and explores a token-based approach to code search.

4

Token-based code indexing

"Sometimes even a truly awesome n-gram might just not been said yet. Just the other day I asked the Google about the sentence 'Batman high-fived Superman' and the Googs found nothing! I DO NOT TRUST ANY MODEL THAT DOUBTS THE GRAMMATICALITY OF THAT SENTENCE."

– T-Rex, in a comic by the Berkeley NLP group[1]

4.1 Introduction

This chapter describes the design of a token-based method for code indexing and search. It first describes the motivation and design goals, followed by an exploration of the design space; finally the chosen technique is evaluated on a real-world data set. As seen in Chapter 3, code reuse is common practice in open source and commercial projects alike, fueled by market pressures, the need for highly complex software and the availability of high-quality reusable software such as the GNU C library and the Linux kernel.

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.

This thesis proposes *search-based decompilation*, a new approach to reverse engineer software. By relying on open-source initiatives such as GNU, the Apache foundation, Linux and BSD distributions, and public code repositories such as Github.com and Google Code (code.google.com), identifying code reuse can be framed as an indexing and search problem.

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 [153]. With this many lines of code on the web today, it is clear that a scalable method for indexing and search is needed. Ideally both speed and accuracy are desirable, so a reasonable approach

[1] `http://nlp.cs.berkeley.edu/Comics.shtml`

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. The former approach is adopted here.

Thus, the primary goal is *performance*: What is needed is a fast information retrieval scheme for executable code. Like a text search engine, having speedy response times is key for usability.

The secondary goals are *precision* (few false positives) and *recall* (few false negatives). Since there is no control over the compiler or the options used, the model has to be robust with respect to compilation and their various optimisations.

This chapter makes the following contributions.

1. The problem of identifying large code bases generated by different compilers and their various optimisations is addressed, which has not been done before.

2. A prototype search engine for binary code called Rendezvous was implemented, which makes use of a combination of instruction mnemonics, control-flow subgraphs and data constants (Section 4.3). Experiments show that Rendezvous is able to achieve a 86.7% F_2 measure (defined in Section 4.7) 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 4.10).

3. As an early prototype, the time-to-index is about 0.4 seconds per function in the worst case, but could be speeded up further in a production system. (Section 4.10.8).

4.2 Design space

Recall the evidence in Chapter 3 that code is largely copied then modified. Since code presented to the retrieval system may be modified from the original, one of the goals is 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.

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 features is not unexplored, and the typical choice to make is between static and dynamic methods. Static analysis was the favoured choice 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 [154]. Static disassembly is by no means a solved problem in the general sense [155], but there are well-known techniques for it, such as linear sweep and recursive traversal [9], and they work well for a wide range of executables.

Three candidate features, or representations, were considered—instruction mnemonics, control-flow subgraphs, 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 subgraph 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 features were chosen as they are derived directly from a disassembly. Instruction mnemonics was chosen as the simplest representation for code semantics; the control-flow graph was chosen as the simplest representation for program structure; data constants were chosen as the simplest representation for data values. A summary of the features considered is given in Table 4.1.

Feature
Instruction mnemonic n-grams [156]
Instruction mnemonic n-perms [157]
Control-flow subgraph [158]
Extended control-flow subgraph
Data constants

Table 4.1: A summary of the different features considered.

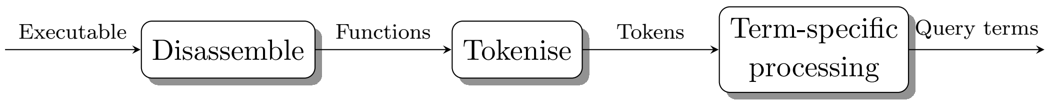


Figure 4.1: Overview of the feature extraction process.

4.3 Feature extraction

The feature extraction procedure involves three steps (Figure 4.1). The executable is first disassembled into its constituent *functions*. The disassembled functions are then *tokenised*, that is, broken down into instruction mnemonics, control-flow subgraphs and constants. Finally, tokens are further processed to form *query terms*, which are then used to construct a search query.

The extraction process for the three different features will now be described in detail using a running example. Figure 4.2 shows the sorting algorithm *bubblesort* written in C.

4.4 Instruction mnemonics

The Dyninst binary instrumentation tool [159] was used 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. The executable is disassembled and the bytes making up each instruction are coalesced together as one

```
// a:  array, n:  size of a
void bubblesort( int *a, int n )
  int i, swapped = 1;
  while( swapped )
    swapped = 0;
    for( i = 0; i < n-1; i++ )
      if( a[i] < a[i+1] )
        int tmp = a[i];
        a[i] = a[i+1];
        a[i+1] = tmp;
        swapped = 1;
```

Figure 4.2: Source code for our running example: bubblesort.

block. Subsequently, the *n-gram* model is used, 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, mnemonics 0 to n 1 are concatenated; to form the second n-gram, mnemonics 1 to n are concatenated and so on. The n-grams are allowed to run over basic block boundaries. A basic block is an instruction sequence that does not contain incoming or outgoing control flow, and usually ends with a jump instruction.

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 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 [157]. 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 4.3.

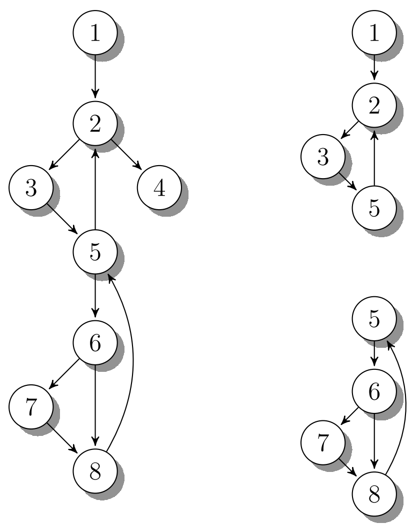
4.5 Control-flow subgraphs

The second feature 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 assumed to

```
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,...`

Figure 4.3: Instruction mnemonics and 1-, 2-, and 3-grams and corresponding n-perms for bubblesort (bottom) based on the first six instructions (top).



	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

Figure 4.4: CFG of bubblesort, two k-graphs, $k = 4$, and their canonical matrix ordering. Each matrix is converted to a 16-bit number by concatenating its rows. Element (0,0) is bit 0; element (3,3) is bit 15. Thus the first subgraph, `1-2-3-5`, corresponds to 0x1214, and the second, `5-6-7-8`, to 0x1286.

return. Conditional instructions, such as `cmov` and `loop` are another exception. The BBs form the nodes in the graph and the flow targets are the directed edges between nodes.

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & V^* \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ V^* & 0 & 1 & 0 & 0 \end{array}$$

$$\begin{array}{c|ccc} & 3 & 5 & 6 \\ \hline 3 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \\ 6 & 0 & 0 & 0 \end{array} \qquad \begin{array}{c|cccc} & 3 & 5 & 6 & V^* \\ \hline 3 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 1 \\ V^* & 1 & 1 & 0 & 0 \end{array}$$

Figure 4.5: 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 subgraph.

The CFG, which can be large, is not indexed as it is. Instead, subgraphs of size k, or k-graphs, are extracted. The approach used is similar to the one adopted by Krügel et al. [158]. Firstly, a list of connected k-graphs are generated from the CFG. 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 subgraph 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 [160], or by brute force since k is small (3 to 7).

Each unique subgraph corresponds to a k^2-bit number. For $k = 4$, a 16-bit value is obtained for each subgraph. Figure 4.4 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 when the rows of the matrix are concatenated. Matrix element (0,0) is the 0th bit and (3,3) the 15th bit.

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, an extension to the k-graph is proposed, called *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 the 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 mentioned before.

4.6 Data constants

The motivation for using constants is the empirical observation that constants do not change with the compiler or compiler optimisation. Two types of constants were con-

```
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’..."
```

Figure 4.6: Data constants for a code snippet from the `usage` function of `vdir`.

sidered: 32-bit integers and strings. Integers considered included immediate operands, displacements and scalar multipliers; the strings considered were ANSI single-byte null-terminated strings.

The extraction algorithm is as follows: all constants are first extracted from an instruction. Explicitly excluded are displacements associated with the stack and frame pointers, or `ESP` and `EBP`, as these depend on the stack layout and hence vary with the compiler, and immediate operands associated with conditional branch instructions.

The constants are then separated by type. Since a 32-bit instruction set is assumed, the data could either be a 32-bit integer or a pointer. An address lookup is made to determine 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, the procedure stops at the first level of indirection. If d_v is a valid ANSI string, that is, it consists of valid ASCII characters and terminated by a null byte, it is assigned type string, otherwise d_v is not used. In all other cases, v is treated as an integer. Figure 4.6 shows the constants extracted from the usage function of the `vdir` program compiled with `gcc` default options.

4.7 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 good 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

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

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, defined as the following.

$$F_\beta = (1+\beta^2)\ \frac{precision \quad recall}{(\beta^2 \quad precision) + recall}$$

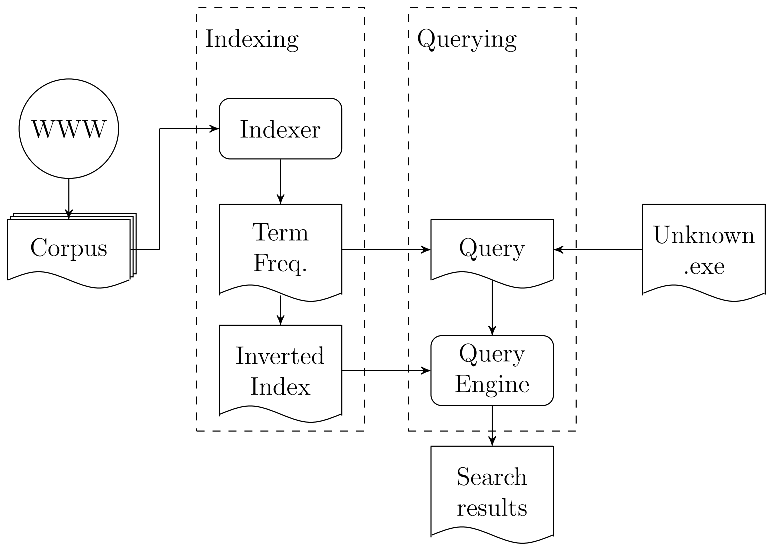


Figure 4.7: Setup for indexing and querying.

When $\beta = 1$, the formula is known as the F_1 measure and both precision and recall are weighted equally. However, the F_2 measure is more favoured:

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

The reason that the F_2 measure is used and not the F_1 measure is that retrieving relevant documents is the main priority; false positives are less of a concern. Thus recall is of higher priority than precision, and F_2 weights recall twice as much as precision.

4.8 Indexing and querying

What we have discussed so far covers the extraction of terms from an executable. In this section, the process of incorporating the tokens into a standard text-based index will be described, along with how queries are made against this index that lead to meaningful search results.

Figure 4.7 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.

Two query models were considered—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. The 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, it is firstly decomposed into a set of terms and encoded as 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, the 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. Three strategies were employed to 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, the first term t_0 is selected and other occurrences of t_0 up to length 2 l_Q removed. This process is repeated until l_Q terms are reached. The $(l_Q + 1)^{th}$ and subsequent terms are truncated.

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

At the end of these steps, what is obtained is a bag of terms for each term category. The terms are firstly concatenated with `AND`, e.g. `XvxFGF AND baNUAL`, which constructs the most restrictive query. 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. This 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, a second query is constructed 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. The document frequency is defined as the number of documents in the corpus in which a term occurs. The terms in S that remain in or are removed from the query are determined by the *document frequency threshold*, $df_{threshold}$. The document frequency threshold is the maximum allowed document frequency for a term to be included in the query. In other words, only terms whose frequency is below $df_{threshold}$ are 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 second `OR` query is to deal with situations where an exact match does not exist, and VSM is relied upon 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.

$$Score(Q, D) = coord(Q, D) \quad C \; \frac{V(Q) \quad V(D)}{V(Q)}$$

where *coord* is a score factor based on the fraction of all query terms that a document contains, C is a normalisation factor, $V(Q) \quad V(D)$ is the dot product of the weighted vectors, and $\quad V(Q) \quad$ is the Euclidean norm. Comparing the scoring function with the cosine similarity measure:

$$cos(\theta) = \frac{V(Q) \quad V(D)}{V(Q) \quad V(D)}$$

the $\quad V(D) \quad$ term is not used on its own in the scoring function 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 [161].

4.9 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 [160] 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 [162] since it is a test of membership. In other words, the Bloom filter consisted of all terms below $df_{threshold}$. The maximum term length was fixed at six, which is sufficient to encode a 32-bit integer using 52 upper- and lower-case letters. Instruction mnemonics were encoded using 8 bits; higher-order bits are truncated. Mnemonic n-grams up to $n = 4$ were considered. Control-flow k-graphs and extended k-graphs were encoded as 32-bit integers. Integer constants were by definition 32-bit integers; strings were truncated to six characters. A total of 10,500 lines of C++ were written for the implementation of disassembly and feature extraction, and 1,000 lines of C++ for indexing and querying.

4.10 Evaluation

This section describes the experiments that were performed to answer the following questions.

What is the optimal value of $df_{threshold}$?

What is the accuracy of the various code features?

What is the effect of compiler optimisation on accuracy?

What is the effect of the compiler on accuracy?

What is the time taken for binary code indexing and querying?

The first evaluation data set used consisted of 2,706 functions from the GNU C library version 2.16 comprising 1.18 million lines of code. The functions were obtained by compiling the library under `gcc -O1` and `gcc -O2` commands, or *GCC1* and *GCC2* respectively. All initialisation and finalisation functions generated by the compiler, such as `_init`, `_fini` and `__i686.get_pc_thunk.bx`, were excluded. The evaluation on this set, referred to as the *glibc* set, consisted of two experiments. In the first experiment, GCC1 is indexed and queries are formed using GCC1 and GCC2. This procedure is then repeated with GCC2 indexed in the second experiment, and the results from both experiments are summed up to obtain the combined precision, recall and F_2 measures.

The second data set was the coreutils 6.10 suite of tools, or the *coreutils* set, which was compiled under `gcc` and `clang` default (-O2) configurations. This data set contained 98 binaries, 1,205 functions, comprising 78,000 lines of code. To obtain the precision, recall and F_2 values one set of functions is similarly indexed and the other used to form queries 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.

The results at a glance is summarised in Table 4.2. The composite model comprising 4-grams, 5-graphs and constants gave the best overall performance for both datasets. This suggests that n-grams, k-graphs and constants are independent, and thus all three are useful as features.

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

Table 4.2: Results at a glance.

4.10.1 Optimal $df_{threshold}$

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

To investigate the optimal value of $df_{threshold}$, an experiment was performed using *glibc* and 4-grams as the term type. The $df_{threshold}$ was varied and the resulting performance measured. The ranked list was restricted to at most five results, that is, if the correct match occurred outside of the top five results, it was treated as a false negative. Table 4.3 shows the results of this experiment. For example, $df_{threshold}$ 1 means that only terms having a document frequency less than or equal to 1 were included, and $df_{threshold}$ means that all terms were included.

Although there is an initial increase from $df_{threshold}$ 1 to $df_{threshold}$ 4, 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 $df_{threshold}$, it was fixed at in all experiments.

$df_{threshold}$	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

Table 4.3: Performance using various values of $df_{threshold}$ on the *glibc* set using 4-grams.

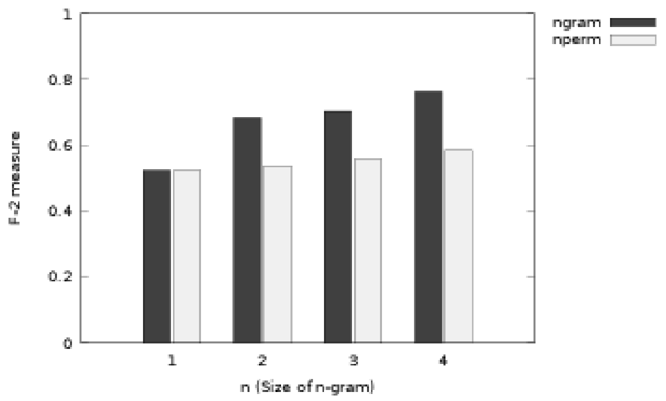


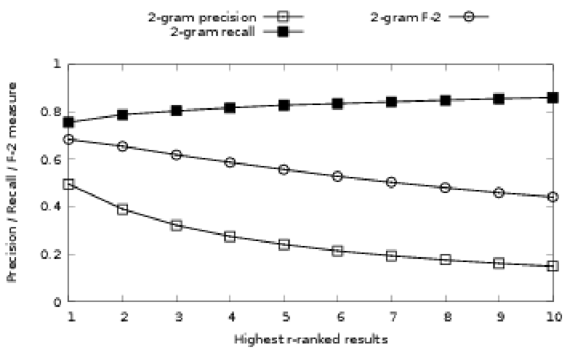
Figure 4.8: The F_2 measures for n-grams and n-perms (glibc data set).

4.10.2 Comparison of n-grams versus n-perms

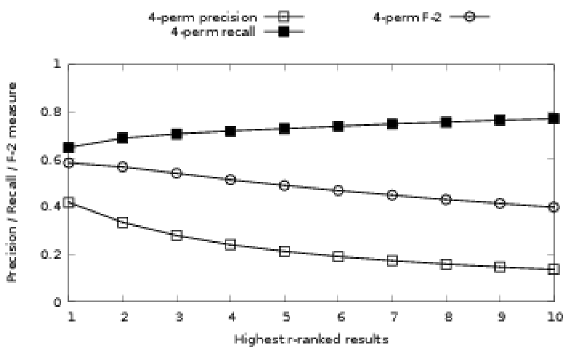
Firstly, the accuracy of n-grams was compared with n-perms, with n taking values from 1 to 4. Values of n larger than 4 were not considered. The *glibc* test set was used to ascertain the accuracy of these two methods in the presence of compiler optimisations. The results are shown in Figure 4.8. The 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 by definition, but the n-gram model out-performed n-perms for $n > 1$. One explanation for this difference was

n	n-gram	n-perm
1	121	121
2	1,483	306
3	6,337	542
4	16,584	889

Table 4.4: The number of unique terms for n-grams and n-perms (*glibc* data set).



(a) 2-gram



(b) 4-perm

Figure 4.9: The precision, recall rates and the F_2 measures for 2-grams and 2-perms of instruction mnemonics (*glibc* data set).

that the n-perm model produced too few terms so that irrelevant results affected recall rates, and this is evident looking at the number of unique terms generated by the two models (Table 4.4). The 2-gram model generated 1,483 unique terms whilst the 4-perm model generated only 889.

Next, 2-grams and 4-perms were analysed in more detail. The precision and recall rates were varied by adjusting the threshold of the r highest ranked results obtained. For example, if this threshold was 1, only the highest ranked result returned by the query engine was considered; the rest of the results were ignored. The value of r was varied from 1 to 10. Figures 4.9a and 4.9b 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 $r = 1$.

There is an observed 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.

4.10.3 Mixed n-gram models

Next, mixed n-gram models were considered to see if combining n-grams produced better results. 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 4.5. 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. 16,584. Also notable was the fact that almost all mixed n-gram models performed better than the single n-gram models.

	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 4.5: Performance of mixed n-gram models by F_2 measure.

	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

Table 4.6: Results for k-graphs and extended k-graphs.

4.10.4 Control-flow k-graphs versus extended k-graphs

In the next set of experiments, control-flow k-graphs and extended k-graphs for $k = 3, 4, 5, 6, 7$ were evaluated. The results are summarised in Table 4.6. The model which gave the highest F_2 was 5-graphs for the *glibc* data set at 0.706, and 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 k-graphs were in fact a better feature than extended k-graphs.

4.10.5 Mixed k-graph models

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

	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 4.7: Results for mixed k-graph models.

	Precision	Recall	F_2
glibc	0.690	0.679	0.681
coreutils	0.867	0.751	0.772

Table 4.8: Results of using data constants to identify functions in the *glibc* and *coreutils* data sets.

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 4.7). Lastly, the mixed k-graph models performed better than the single k-graph models.

4.10.6 Data constants

Table 4.8 shows the results of using the third feature, 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* contained strings, whilst 889 functions, or 40.3% of functions in the *coreutils* did.

4.10.7 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 extracted then concatenated to form a composite document for each function.

Two composite models were considered; 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, the models for the *glibc* set that were tested comprised 4-gram/5-graph/constants and 1-gram/4-gram/3-graph/5-graph/constants. 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 listed in Table 4.9.

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 three components fared better than the model with five.

In addition to maximising F_2, the recall rates for $r = 10$ were of interest, since users of the search engine are not expected to venture beyond the first page of results. Considering only the top ten 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, 206 were found to be 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 4.10). The compiled executables, or `getfsent01` and `getfsent02` respectively, differed significantly due to several factors. Firstly, the function `fstab_fetch` was inlined, causing the `mov` and `call` instructions in `getfsent01` to be expanded to eight.

Secondly, there were two instruction substitutions: instruction `mov eax, 0x0` in `getfsent-01` was substituted by the `xor eax, eax` instruction which utilises two bytes instead of five; 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 `getfsent02`.

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 [163]. This is left to future work.

```
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);
```

Figure 4.10: Source code of `getfsent`

		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

Table 4.9: Results of the composite models. Where indicated, variable r is the number of ranked results considered, otherwise $r = 1$.

		Average (s)	Worst (s)
Feature	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 (2,410 functions)		907.448	981.081
Total per function		0.377	0.407

Table 4.10: Average and worst-case timings for *coreutils* set.

4.10.8 Timing

The final set of experiments was performed to determine the time taken for a binary program to be disassembled, for the terms to be extracted and for the query to return with the search results. The *coreutils* set was timed for this experiment, and the timing included both the `gcc`-compiled code and the `clang`-compiled code to give a total of 2,410 functions. Table 4.10 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 feature extraction. The total time was computed by summing the time taken to extract each of the three features. Strictly speaking, this time is an overestimate since the binary was disassembled two more times than was necessary in practice. On the other hand, no end-to-end timings were done to take into consideration the computational time required for the front-end system, so the timings are an approximation at best.

It was 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 there are plans to streamline this procedure in future.

4.11 Discussion

4.11.1 Limitations

An important assumption made in this chapter is that the binary code in question is not deliberately obfuscated. The presence of code packing, encryption or self-modifying code would make disassembly, and therefore feature extraction, 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, static analysis was considered primarily for its performance. Including dynamic methods in Rendezvous is left to future work.

4.11.2 Threats to validity

Threats to internal validity include the limited software tested, and the limited number of compilers and compiler optimisation levels used. The high accuracy observed 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. Optimisation level `gcc -O3` was not considered.

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.

4.11.3 Mnemonic n-grams and basic block boundaries

In this chapter, n-grams were allowed to run over BB boundaries. The implication of this is that functions which have short BBs, e.g. three instructions or less, might not be correctly detected using 4-grams alone, especially if there is BB reordering during compiler optimisation. This could explain why 1+4-grams performed better than the 4-gram model. The alternative approach is to disallow n-grams from running over BB boundaries. However, this comes with the penalty of having fewer unique n-grams available. Further experiments are needed to determine if this alternative approach does indeed perform better.

4.12 Related work

The line of work that is most closely related is that of binary code clone detection. Sæbjørnsen et al. [151] 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 code clone matching across different code bases, it was necessary to address the compiler optimisation problem and the proposed technique has been demonstrated to be sufficiently accurate to be successful. Hemel et al. [152] 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 the experiments show, between 60–70% of functions were identified using only string constants. Other approaches include directed acyclic graphs [156], program dependence graphs [164] and program expression graphs [165]. None of these approaches were considered 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 [166]. Examples include CCFinder [167], CP-Miner [168], MUDABlue [169], CLAN [170] and XIAO [171]. 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 features that have been studied include byte values [172], opcodes [173], control-flow subgraphs [158], call graphs [150], as well as run-time behavioural techniques [174]. Even though Rendezvous borrows techniques from this field, the aim is to do more fine-grained analysis, and identify binary code at a function level. At the moment, only code obfuscation up to the level of the compiler and its different optimisations is considered.

5

Perturbation analysis

"Does the flap of a butterfly's wings in Brazil set off a tornado in Texas?"
– Title of chaos theory pioneer Edward Lorenz's 1972 talk

Search-based decompilation is premised on being able to match machine code with its functional equivalent in a reference corpus. Token-based matching (Chapter 4) is one such approach to accomplish this task. However, a problem arises when different compilers or compiling options are used, resulting in the failure to match, say, syntactically different instruction sequences, such as `xor EAX, EAX` and `mov EAX, 0`, or to deal with basic block reordering.

Performing symbolic execution with theorem proving as advocated by Gao et al. [175] in a tool called BinHunt is one possible solution. BinHunt first tries matching basic blocks for semantic equivalence by generating the symbolic expressions for all basic-block output variables. Since the compiler may use different registers and variables, BinHunt tries all possible permutations of symbolic expressions to obtain a match, if one exists. The *matching strength* of two basic blocks is defined to be 1.0 if the matching permutation uses the same registers; otherwise the matching strength is assigned the value 0.9 if different registers are used. This is to deal with context mismatch. The second mechanism implemented by BinHunt is backtracking-based subgraph isomorphism that relies on the results of basic-block matching.

However, BinHunt does not differentiate between different implementations of the same algorithm, e.g. bubblesort and quicksort, or independently developed code not due to copy and paste. Performing *structural analysis* is important for search-based decompilation, since one hardly expects an implementation of bubblesort to decompile to quicksort, or vice versa. Existing approaches to binary code clone detection assume that the underlying application is closely related [149, 150, 175], or that the clone is a result of copy and paste [176, 152].

The rest of this chapter makes the following contributions.

This chapter addresses the problem of differentiating between different implementations of the same algorithm, which has not been addressed before.

This chapter describes *perturbation analysis*, a technique that can identify structural similarity. It is able to distinguish between 11 subtle variants of 5 sorting algorithms (Section 5.3).

```
f1:     mov  ECX, EAX     g1:     int g(int x)
f2:     add  ECX, 2       g2:
f3:     imul EAX, ECX     g3:        return x*x + 2*x + 1;
f4:     add  EAX, 1       g4:
f5:     retn
```

Figure 5.1: Our running example. Assembly program f, which computes $x(x+2)+1$, is the compiled version of the C program g, which computes x^2+2x+1.

A source-binary indexing scheme based on test-based indexing (TBI, Section 5.4) and perturbation-based indexing (PBI, Section 5.8) is proposed. Experiments conducted on 100 functions in `coreutils` 6.10, compiled with 5 different configurations (`gcc -O2`, `gcc -O3`, `gcc -Os`, `clang -O2` and `clang -O3`), show that the total precision and recall achieved is 68.7% and 95.2% respectively for TBI and PBI combined (Section 5.10.4).

5.1 Overview

Consider the equivalent functions, f and g, in Figure 5.1 and their corresponding listings (`gcc` default options were used). The argument x is stored in register `EAX` at the start of f. It is not immediately obvious that the two functions are semantically equivalent. Indeed, different compilation optimisations may, and often do, yield significant syntactic differences in the binary. The goal is to match the source code of g to that of the binary code of f.

There are at least three possible approaches to accomplish this. The first method is to compile the source code down to binary form and do the comparison at the machine code level. The second method is to decompile g up to the higher-level language of f. A third approach is to do both—compile f and decompile g to a suitable intermediate representation. The second approach was chosen since, as a programming language, C has been, and continues to be, widely used in the compiler, decompiler and software testing communities. This is evident from the fact that tools continue to be written for it, many more than those targeting binaries and intermediate representations. For the purposes of this chapter, it is assumed that the source code to match against is in C, since the tools used are written to analyse C, as are many of the functions in the reference corpus, including the GNU C library and the `coreutils` tool set.

The first step in the process is obtaining a disassembled version of the binary. The disassembly is then lifted to C via an assembly-to-C translator. Two source-binary matching techniques are used: test-based indexing (TBI) and perturbation-based indexing (PBI). TBI tests two functions for input-output congruence (Section 5.4); PBI, which makes use of perturbation analysis (Section 5.6), tests two functions for structural similarity (Section 5.8). We first discuss the implementation of the assembly-to-C translator.

```
unsigned int eax;
bool cf, pf, af, zf, sf, of;
...
cf = (eax + 2) < eax;
pf = parity_lookup[ (char)(eax + 2) ];
af = (((char)eax ^ 2 ^ (char)(eax + 2) )
     & 0x10) != 0;
zf = (eax + 2) == 0;
sf = (eax + 2) >> 31;
of = ( ~(eax ^ 2) & (2 ^ (eax + 2))
     & 0x80000000 ) != 0;
eax = eax + 2;
```

Figure 5.2: The assembly-to-C translation of `add eax, 2`.

5.2 Assembly-to-C translation

The aim of doing assembly-to-C translation is solely to express the semantics of the disassembled instructions in valid C syntax, since analysis is done in C. Readability is not a concern here, so it is sufficient to perform direct per-instruction translation. The CPU registers, flags and the stack are treated as local variables. A 32-bit Intel x86 CPU is assumed, and each instruction in the function is decoded to determine three things: the operands, the operating width, i.e. 8-, 16- or 32-bits, and the operations to perform on the simulated CPU and stack. Memory locations are dereferenced as indicated by the operating width. For example, `mov AL, [ECX]` is translated as `al = *(char *)ecx;`.

The output of the translation is a sequence of C statements implementing the semantics one instruction at a time. For example, `add EAX, 2` is translated as shown in Figure 5.2. From the translation, we can see that the carry flag (`cf`) is set when an unsigned overflow occurs, while the overflow flag (`of`) indicates a signed overflow when `eax = 0x7FFFFFFE`.

The layout of the basic blocks in the disassembled function is preserved in the C version. Jumps in the program are translated as `goto` statements; calls are implemented as per calls in C syntax, with the appropriate registers or stack locations as input parameters and return variables. The target of indirect jumps and calls, e.g. `jmp [EAX]`, need to be determined before translation, and at the moment this is done manually. External calls to libraries and system calls are currently unsupported.

5.3 Source-binary matching for functions

Before describing the algorithm for source-binary matching, we briefly define some notation.

Definition (State) A *state*, s, is represented by the tuple

$$s = a_1, \ldots, a_m, x_1, \ldots, x_n, v$$

where a_i are the input parameters, x_j are the program variables, including stack and global variables, and v is the output variable. A valid state s is one which belongs to a global finite non-empty set of states . At the initial state, s_0, only a_i are defined. At a final state, s_f, v is defined.

Definition (Program) A *program* is a finite set of pairs of the form (C_i, T_i) where i is the location and C_i is the computation to perform on the current state s_{i-1} to generate a new state $s_i = C_i(s_{i-1})$. T_i is a predicate that determines (C_{i+1}, T_{i+1}) based on s_i.

Definition (Path) A *path* is a finite sequence of (C_i, T_i) pairs constituting a legal path through the program applying successive transformations to the program state. For example,

$$s_k = C_k \quad C_{k-1} \ldots C_2 \quad C_1(s_0)$$

where denotes the composition of functions.

For brevity only the output variable defined in the post-computation state s_k is listed.

Definition (Congruence) For two procedures f and g, we say f and g are *congruent* if all terminating paths of f and g starting from the same global input state s_0, return the same global output state $s_f^f = s_f^g$.

5.4 Test-based indexing

The problem of determining congruence can be reduced to one of code coverage testing: simply call the two procedures with equal inputs, and assert that they return the same output. Existing test cases can provide this input. Alternatively, automated software testing tools [177, 178, 179] can help.

A possible third approach is to perform random testing in what is termed in this chapter as *test-based indexing*. The main premise is that given random input, a function should produce an output that is unique regardless of the implementation.

The first step is to infer the prototype of a function f. It is assumed that the function prototype is known. In practice, there are methods to perform function prototyping, such as through reaching definitions [180] and liveness analysis [11]. The prototype determines its test harness, which is used to initialise the state of a function with a pre-determined pseudo-random input I, and retrieve the outputs O. The test harness is itself a C program that calls f as a sub-routine. The arguments are initialised according to their type. For instance, a `char` is assigned a random 8-bit number, a `short` a random 16-bit number and so on; pointer-types such as `char *` and `char **` are assigned a pointer to the appropriately populated data structure. The `void *` type is ambiguous, and is treated as a `char *`. The compound data type, or a `struct`, is initialised according to their constituent types. Similarly, the return type is treated according to its type.

Secondly, $f(I)$ is executed with the help of the test harness to obtain O. To produce a test-based index, an I, O pair is hashed to give $H(I, O)$. The procedure for retrieval is then a matter of matching hashes with the reference index, which can be done efficiently. We define the variable n_{TBI} which determines the number of

	f	$s_0^f = \langle eax_0 = a\rangle$		h	$s_0^h = \langle x_0 = a\rangle$
f1:	`mov ECX, EAX`	$s_1^f = \langle ecx_1 = a\rangle$	h1:	`x = x*(x+2);`	$\boldsymbol{s_1^h = \langle x_1 = a(a+2)\rangle}$
f2:	`add EAX, 2`	$s_2^f = \langle eax_2 = a+2\rangle$	h2:	`x++;`	$s_2^h = \langle x_2 = a(a+2)+1\rangle$
f3:	`imul EAX, ECX`	$\boldsymbol{s_3^f = \langle eax_3 = a(a+2)\rangle}$	h3:	`return x;`	$s_f^h = \langle v = a(a+2)+1\rangle$
f4:	`add EAX, 1`	$s_4^f = \langle eax_4 = a(a+2)+1\rangle$			
f5:	`retn`	$s_f^f = \langle v = a(a+2)+1\rangle$		k	$s_0^k = \langle x_0 = a\rangle$
			k1:	`y = 1;`	$s_1^k = \langle y_1 = 1\rangle$
	g	$s_0^g = \langle x_0 = a\rangle$	k2:	`y += x*(x+2);`	$s_2^k = \langle y_2 = a(a+2)+1\rangle$
g1:	`x++;`	$s_1^g = \langle x_1 = a+1\rangle$	k3:	`return y;`	$s_f^k = \langle v = a(a+2)+1\rangle$
g2:	`return x*x;`	$s_f^g = \langle v = (a+1)*(a+1)\rangle$			

Figure 5.3: Functions f, g, h and k are congruent as they have common input and output states. Functions f and h share a common intermediate state at (s_3^f, s_1^h). However, h and k do not share any intermediate states even though they comprise the addition of otherwise identical terms.

$$H(\quad I, Q_1\quad), H(\quad I, Q_2\quad), \ldots, H(\quad I, Q_{n_{TBI}}\quad)$$

hashes to use as an index.

5.5 Identifying structural similarity

The problem, however, is that the above technique is external and nothing is known about the internal algorithm of the procedure. For instance, under this matching scheme, if f was bubblesort and g was mergesort, a reasonable conclusion is that the procedure was *a* sorting function, nothing more. This is insufficient if the goal of the analyst is to retrieve the algorithm from the binary. Moreover, in the case where two or more congruent matches are found, a way of further narrowing the search is clearly needed. In short, a higher-fidelity matching scheme is called for.

A simple approach to include more internal distinction, i.e. identify *structural similarity*, is to locate common intermediate states. This can be done by tracking the states and comparing them path by path. We revisit our running example of Figure 5.1, and have annotated it with state information in Figure 5.3. The states are expressed in terms of the input parameter, a. We observe that since function f shares more common states with h, we say that f is structurally more similar to h than g.

Determining structural similarity using intermediate state, however, is sensitive to instruction reordering. Suppose that the addition of the two terms in function h were reordered as in function k in Figure 5.3. Function k does not share any common intermediate states with f even though their algorithms comprise the addition of otherwise identical terms.

5.6 Perturbation analysis

In this section, a better structural comparison algorithm is presented, called *perturbation analysis*. A *perturbation function* P : is a localised state transformation that given a state s returns $P(s)$. A perturbation is localised in the sense that the transformation is applied to a single variable in a state. If unambiguous, $P(s_i)$ is taken to mean P is applied to the variable used or defined in state i, otherwise an index is added,

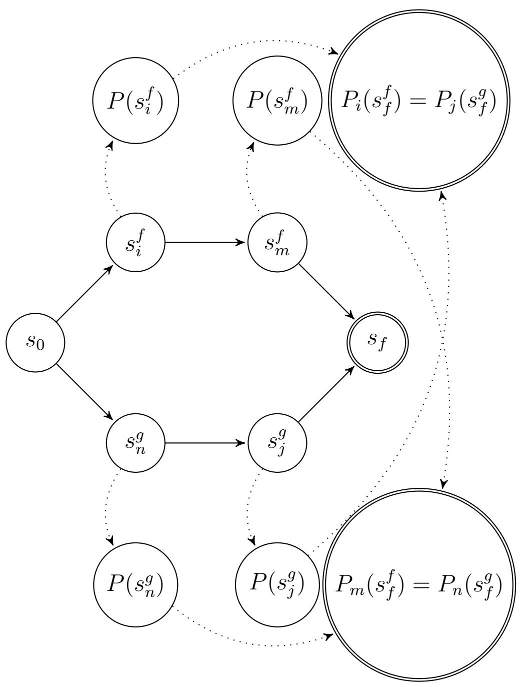


Figure 5.4: An illustration of two congruent procedures f and g perturbed at states i, j, m and n, resulting in new final states for which $P_i(s_f^f) = P_j(s_f^g)$ and $P_m(s_f^f) = P_n(s_f^g)$.

e.g. $P(s_{1a})$. The intuition is that perturbation induces a sufficiently distinct final state, and two structurally similar procedures should, for some variable pair, result in the same deviant state. The more such variable pairs there are, the more structurally similar the procedures. This may be thought of as an analogue of differential fault analysis [181, 182]. A more formal definition is as follows.

Definition (State congruence) Given two congruent procedures f and g, and a perturbation function P that is applied at state i in f resulting in the final state $P_i(s_f^f)$ such that $P_i(s_f^f) \neq s_f^f$, and there exists a state j in g that results in the final state $P_j(s_f^g)$ such that $P_i(s_f^f) = P_j(s_f^g)$, then we say that states i and j are congruent in P. Alternatively, we say that i and j are in a congruent set of P. The expression $P_i(s_f)$ is used to refer to the final state when P is applied to s_i, that is

$$P_i(s_f) = C_n \quad C_{n-1} \quad \ldots \quad C_{i+1} \quad P \quad C_i \ldots C_1(s_0)$$

Figure 5.4 shows an example of procedures f and g that are congruent as they have the same final state from a common input state. In addition, when P is applied to states i in f and j in g, they reach the same deviant output state denoted by $P_i(s_f^f) = P_j(s_f^g)$. Thus, states i and j are congruent in P. Similarly, states m and n are congruent in P. Note also that i and j are in a different congruent set as m and n as the final states are distinct.

An example of a perturbation function is the definition-increment operator, $P^{def}(\; x \;) = \; x + 1 \;$. Applying $P^{def}(s_i)$ is taken to mean "post-write increment the variable defined at s_i". Applying this to our example, $P^{def}(s_4^f)$ results in the final state (in terms of the input parameter a)

$P_1^{def}(s_f^f) = \langle v = (a+1)(a+2)+1 \rangle$
$P_2^{def}(s_f^f) = \langle v = a(a+3)+1 \rangle$
$\boldsymbol{P_3^{def}(s_f^f) = \langle v = a(a+2)+2 \rangle}$
$\boldsymbol{P_4^{def}(s_f^f) = \langle v = a(a+2)+2 \rangle}$
$P_1^{def}(s_f^g) = \langle v = (a+2)*(a+2) \rangle$
$\boldsymbol{P_1^{def}(s_f^h) = \langle v = a(a+2)+2 \rangle}$
$\boldsymbol{P_2^{def}(s_f^h) = \langle v = a(a+2)+2 \rangle}$
$\boldsymbol{P_1^{def}(s_f^k) = \langle v = a(a+2)+2 \rangle}$
$\boldsymbol{P_2^{def}(s_f^k) = \langle v = a(a+2)+2 \rangle}$

(a) P^{def}

	Congruence set
$\boldsymbol{P_2^{use}(s_f^f) = \langle v = a(a+3)+1 \rangle}$	**1**
$\boldsymbol{P_{3a}^{use}(s_f^f) = \langle v = a(a+3)+1 \rangle}$	**1**
$\boldsymbol{P_{3b}^{use}(s_f^f) = \langle v = (a+1)(a+2)+1 \rangle}$	**2**
$\boldsymbol{P_4^{use}(s_f^f) = \langle v = a(a+2)+2 \rangle}$	**3**
$P_{2a}^{use}(s_f^g) = \langle v = (a+2)(a+1) \rangle$	
$P_{2b}^{use}(s_f^g) = \langle v = (a+1)(a+2) \rangle$	
$\boldsymbol{P_{1a}^{use}(s_f^h) = \langle v = (a+1)(a+2)+1 \rangle}$	**2**
$\boldsymbol{P_{1b}^{use}(s_f^h) = \langle v = a(a+3)+1 \rangle}$	**1**
$\boldsymbol{P_2^{use}(s_f^h) = \langle v = a(a+2)+2 \rangle}$	**3**
$\boldsymbol{P_1^{use}(s_f^k) = \langle v = a(a+2)+2 \rangle}$	**3**
$\boldsymbol{P_{2a}^{use}(s_f^k) = \langle v = (a+1)(a+2)+1 \rangle}$	**2**
$\boldsymbol{P_{2b}^{use}(s_f^k) = \langle v = a(a+3)+1 \rangle}$	**1**

(b) P^{use}

Figure 5.5: a. Perturbation function $P^{def}(\ x\ ) = \ x+1$ applied to our running example. Instructions `f3`, `f4`, `h1`, `h2`, `k1` and `k2` are congruent in P^{def}. b. Perturbation function P^{use} applied to our running example produces three congruent sets: `f2`, `f3`, `h1`, `k2` , `f3`, `h1`, `k2` and `f4`, `h2`, `k1` , corresponding to the two x terms and the constant 1 term.

$$P_4^{def}(s_f^f) = \quad v = a(a+2)+2$$

The same perturbation applied to $P^{def}(s_1^k)$ and $P^{def}(s_2^h)$ produces a matching final state

$$P_1^{def}(s_f^k) = P_2^{def}(s_f^h) = P_4^{def}(s_f^f) = \quad v = a(a+2)+2$$

Extending this to the rest of the states, we obtain the congruent set under P^{def} comprising `f3`, `f4`, `h1`, `h2`, `k1` and `k2`. On the other hand,

$$P_1^{def}(s_f^g) = \quad v = (a+2) \quad (a+2) \qquad = \quad v = a(a+2)+2$$

implying that instruction `g1` is not in that set. The other possible final states $P_i^{add}(s_f)$ are shown in Figure 5.5a.

The second perturbation function is the use-increment operator, P^{use}, which is similar to P^{def}, except that P^{use} operates on variable uses instead of definitions. Informally, $P^{use}(s_{ik})$ can be described as "pre-read increment the k^{th} variable used in state i". A subscript is appended to the state index as a state may use more than one variable. If P^{use} is applied to our example again, the final states in Figure 5.5b are obtained.

Undefined behaviour may occur when the domain of a transformed variable x, written $dom(x')$, is undefined. Since our example consists solely of integers it is easy to see that if $dom(x)$ is defined, then $dom(x')$ is also defined. However, if x were to be used as an index into an array, then the validity of $dom(x')$ depends on whether it exceeds the array bounds.

5.7 Guard functions

Perturbation analysis is performed by instrumenting, that is, adding additional instructions to, all possible perturbation sites in a function, one per variable of interest according to the perturbation function P. The program is modified to add a guard input parameter *guard* which is of `int` type, and each site is guarded with a guard function $G_i : int \quad bool$ that activates the transformation P_i if it evaluates to `true`. The function G^{single}, which activates one guard at a time, is defined as

$$G_i^{single}(guard) = \begin{cases} true, & guard = i \\ false, & otherwise \end{cases}$$

A second guard function is defined for when more than one active site is desired. The following guard function, $G^{bitvector}$, allows this.

$$G_i^{bitvector}(guard) = \begin{cases} true, & (guard >> i)\&1 = 1 \\ false, & otherwise \end{cases}$$

Guard $G^{bitvector}$ allows the matching of variables that are split or merged as a result of compilation, for example in partial loop unrolling optimisations.

Given an existing test suite, the congruent sets can be automatically derived by iterating through all values of the guard variable and solving for $f_G(x, guard) = h_G(x, guard)$ where f_G and h_G are the guarded versions of f and h, and x are the test cases.

Solutions are of the form $(guard_i, guard_j)$ which, if G^{single} is used, imply the pair of sites (i, j) are congruent. The set of congruent pairs for our example in Figure 5.5b is $\quad (3,5), (2,2), (1,3) \quad$. The Jaccard coefficient is used to measure the structural similarity between f and h. Let $S^f \quad S^h$ be the set of congruent pairs and $S^f \quad S^h$ be the set of perturbation sites, then the structural similarity of f and h is given by

$$SS(S^f, S^h) = \frac{S^f \quad S^h}{S^f \quad S^h} = \frac{6}{6} = 100\%$$

5.8 Perturbation-based indexing

Perturbation-based indexing (PBI) follows from perturbation analysis and is similar to test-based indexing (TBI), except that f is transformed to its guarded form, f_G. Output O_S, which consists of an output state for each active perturbation site in S, is then obtained by executing $f_G(I, guard)$. As with TBI, n_{PBI} is defined to be the number of hashed input-output pairs $H(\ I, O_{s_1} \ ), \ldots, H(\ I, O_{s_{n_{PBI}}} \ )$ to be included in the index. Because the search space of variables whilst performing perturbation analysis can be large, the number of perturbation sites explored is restricted by n_{PBI}.

To evaluate TBI and PBI, the definitions of *precision*, *recall* and the F_2-measure in Section 4.7 are reused.

5.9 Implementation

Disassembly was performed using the Dyninst binary instrumentation framework.The assembly-to-C translator was based on the Bochs CPU emulator [183] and was implemented as a Dyninst plugin in 1,581 lines of C. The compile-time library that implements

		$G^{bitvector}$				G^{single}			
		P^{def}		P^{use}		P^{def}		P^{use}	
B	vs B	122/143	(85.3%)	145/167	(86.8%)	8/8	(100%)	12/12	(100%)
B	vs I	46/129	(35.7%)	36/179	(20.1%)	4/9	(44.4%)	4/13	(30.8%)
B	vs S	0/129	(0%)	0/167	(0%)	0/9	(0%)	0/13	(0%)
I	vs B	63/143	(44.1%)	58/162	(35.8%)	4/7	(57.1%)	4/11	(36.4%)
I	vs I	113/129	(87.6%)	147/174	(84.5%)	8/8	(100%)	12/12	(100%)
I	vs S	0/129	(0%)	0/162	(0%)	0/8	(0%)	0/12	(0%)
S	vs B	0/143	(0%)	0/162	(0%)	0/8	(0%)	0/11	(0%)
S	vs I	0/129	(0%)	0/174	(0%)	0/9	(0%)	0/12	(0%)
S	vs S	128/129	(99.2%)	155/162	(95.7%)	9/9	(100%)	12/12	(100%)

Table 5.1: Structural similarity between three sorting algorithms bubblesort (B), insertionsort (I) and selectionsort (S). The comparison is between source and the binary compiled with `gcc -O0` using the perturbation functions P^{use} and P^{def}, and the two guard functions G^{single} and $G^{bitvector}$.

the x86 instruction semantics was written in 1,192 lines of C. Function prototypes were obtained using the output of `gcc -aux-info`. All experiments were carried out on an Intel Core 2 Duo machine running Ubuntu 12.04 with 1 GB of RAM.

5.10 Evaluation

The most important questions appear to be the following.

1. What is the accuracy of perturbation functions P^{def}, P^{use} and guard functions G^{single},$G^{bitvector}$?
2. Do compiler optimisations or the compiler affect accuracy?
3. What are the optimal values for the number of test-based and perturbation-based indices, or n_{TBI} and n_{PBI} respectively?
4. What is the accuracy of test-based and perturbation-based indexing?

Two data sets were used in the evaluation. The first set, or the *sorting* set, comprised five well-known sorting algorithms – bubblesort, insertionsort, selectionsort and quicksort. The algorithms as listed in Wikipedia.org and the Rosetta.org code repository were used. The second data set consisted of 100 internal functions, and comprising 172,648 instructions, from `coreutils 6.10`, also called the *coreutils* set. These functions were chosen as they did not make system or API calls.

5.10.1 Perturbation and guard functions

As a first test, structural similarity analysis was performed on three sorting algorithms: bubblesort, insertionsort and selectionsort. The functions were compiled with `gcc -O0`, then translated to C and instrumented with the two perturbation functions, P^{def} and P^{use}, using the two guard functions, G^{single} and $G^{bitvector}$, giving a total of four possible combinations. A randomly initialised array of 100 integers was used as input, and the

	R-B	R-BO	R-I
W-BO	**4/8 (50%)**	**4/8 (50%)**	2/7 (28.6%)
W-I	2/7 (28.6%)	2/7 (28.6%)	**6/6 (100%)**

(a) Bubblesort/Insertionsort

	W-Half	W-First	W-Last	R-Half	R-First	R-Last
W-Half	16/16 (100%)	-	-	-	-	-
W-First	0/16 (0%)	16/16 (100%)	-	-	-	-
W-Last	0/16 (0%)	0/16 (0%)	16/16 (100%)	-	-	-
R-Half	0/13 (0%)	0/13 (0%)	0/13 (0%)	10/10 (100%)	-	-
R-First	0/13 (0%)	0/13 (0%)	0/13 (0%)	0/10 (0%)	10/10 (100%)	-
R-Last	0/13 (0%)	0/13 (0%)	0/13 (0%)	0/10 (0%)	2/10 (20%)	10/10 (100%)

(b) Quicksort

Figure 5.6: **a.** Structural similarity between Wikipedia-Bubblesort (W-B), Wikipedia-Bubblesort-optimised (W-BO), Wikipedia-Insertionsort (W-I), Rosetta-Bubblesort-optimised (R-BO), and Rosetta-Insertionsort (R-I). **b.** Structural similarity of quicksort with different pivot points for the Wikipedia and Rosetta implementations. Half - Pivot is the $(n/2)^{th}$ element, First - Pivot is the 0^{th} element, Last - Pivot is the $(n \quad 1)^{th}$ element.

structural similarity between the three functions was measured using the method described in Section 5.7. The results are listed in Table 5.1.

If the threshold for structural similarity was set at 50%, all combinations except for G^{single} and P^{def} had a 100% detection rate. Looking at the results for $G^{bitvector}$, the similarity was never 100% when it should have been. The reason was because $G^{bitvector}$ allows all possible combinations of perturbations, some of which could not be matched.

It was also observed that the similarity between bubblesort and insertionsort obtained using $G^{bitvector}$ was not 0%. The spurious match was found to be due to the "swap block" in bubblesort and the innermost `a[iHole] = a[iHole-1]` assignment of insertionsort. Because this assignment was executed the same number of times as the swap block in bubblesort, they were found to be congruent.

To conclude, P^{use} performed better than P^{def}. Apart from the results, $G^{bitvector}$ has a severe disadvantage in the number of perturbation sites it can accommodate, which in this case is 32. In contrast, G^{single} is able to accommodate up to 2^{32} sites. These two observations motivated the decision to use G^{single} and P^{use} for the rest of the experiments.

5.10.2 Comparison of different implementations

The aim of the next set of experiments was to test the ability of perturbation analysis to differentiate between implementations of the same algorithm, in this case the implementations of bubblesort and insertionsort from Wikipedia (W-BO, W-I) as well as the Rosetta.org code repository (R-B, R-BO, R-I). The difference between R-B and R-BO was: the number of iterations of the inner loop decreased with each successive iteration of the outer loop of R-BO. In addition, the quicksort algorithm was varied by tuning it to use different pivots (W-Half, W-First, W-Last, R-Half, R-First and R-Last).

Perturbation analysis found no difference between the insertionsort implementations (Table 5.6a). On manual inspection, the only noticeable difference was the starting index of the outer loops; the outer loop of the Rosetta algorithm started from index 1, and

n_{TBI}	Precision	Recall
1	105/169 (62.1%)	105/105 (100%)
2	105/165 (63.6%)	105/105 (100%)
5	105/165 (63.6%)	105/105 (100%)
10	105/165 (63.6%)	105/105 (100%)
100	105/165 (63.6%)	105/105 (100%)

Table 5.2: The precision and recall rates for self-test `gcc -O2` versus `gcc -O2`, using TBI with $n_{TBI} = 1, 2, 5, 10, 100$, the number of input-output pairs.

that of Wikipedia started from index 0. Not surprisingly, when compared with the bubblesort algorithms, the Rosetta insertionsort gave the same result as the Wikipedia one. The Rosetta bubblesort (R-BO), however, was not the same as its Wikipedia counterpart (W-BO), due to the outer loop. While the number of iterations of the R-BO inner loop decreased by one at each successive outer loop iteration, the W-BO algorithm was more optimised, with each successive inner loop limit determined by the last swap encountered in the previous iteration. Despite this difference, the total number of executions of the swap block was the same amongst all bubblesort algorithms. As a result, perturbation analysis still found their similarity to be 50%.

In contrast, perturbation analysis was not able to find any similarity between the same quicksort implementation given three different initial pivots, resulting in almost no matches (Table 5.6b). The choice of pivot affected the dynamic behaviour of the algorithm too much. As to the different quicksort implementations, the Wikipedia-Quicksort (W-Q) partitions the array into two with a single pointer in an ascending for-loop, swapping elements less than the pivot towards the bottom indices. The Rosetta-Quicksort (R-Q) algorithm does the partitioning using two pointers in a two-nested while loop, one pointer starting from index 0, the other from index $n \quad 1$, and completing the partition when the two pointers cross each other. Moreover, the W-Q algorithm had three swap blocks, while R-Q had only one. As a result the number of perturbation sites was less in the R-Q case—5 versus 8—and perturbation analysis could not find any similarity between the two implementations.

5.10.3 *Coreutils* data set

In this set of experiments, the accuracy of test-based indexing as described in Section 5.4 was evaluated. The first step was to study the effect of n_{TBI}, the number of TBI input-output pairs, on performance. Table 5.2 shows that the precision and recall increased as n_{TBI} increased, peaking at $n_{TBI} = 2$. The maximum precision and recall rates obtained for TBI was 63.6% and 100% respectively. TBI generated 84 unique MD5 hashes for the 105 functions analysed.

Keeping n_{TBI} constant at 10, a further experiment was performed to study the effect of including n_{PBI}, the number of perturbation-based input-output pairs. The value of n_{PBI} which gave the best performance was found to be 1, with precision and recall at 70.9% and 100% respectively (Table 5.7a).

The largest group giving the same input-output hash consisted of a group of four functions dealing with comparisons of data structures (Table 5.7b). On manual analysis, it was discovered that `ls.c:dev_ino_compare` and `du.c:entry_compare` were in fact exact copies of each other. The reason for the unexpected match between functions `fts-cycle.c:AD_compare` and `src_to_dest_compare` was: both functions dealt with data structures that happened to have the same pointer offsets.

5.10.4 Compilers and compiler options

The aim of this set of experiments was to test the accuracy of TBI and PBI to detect the *coreutils* functions compiled with two compilers, `gcc` and `clang`, using different compile options. The results are shown in Table 5.3.

The first observation was that TBI, or $n_{PBI} = 0$, was out-performed by TBI/PBI in all data sets. The F_2 measure of PBI peaked at $n_{PBI} = 2$ for `gcc` and overall. This was surprising since the performance was expected to improve with larger values of n_{PBI}, and with more perturbation sites in use. One possible explanation is that certain compiler optimisations created variables that had unique perturbation behaviour. There was tension observed between precision and recall, especially in the two sets of `clang`-compiled functions; higher precision was accompanied by lower recall, and vice-versa.

False negatives were partly due to "function clones"—variants of a frequently-called routine that the compiler creates to optimise run-time performance. As a result of cloning, the parameters and their number can change, so that the function prototype given by `gcc -aux-info` no longer matches and thus the input state was not initialised correctly. Other false negatives were due to invalid pointer dereferences, resulting in undefined behaviour.

n_{PBI}	Precision	Recall
1	105/148 (70.9%)	105/105 (100%)
2	105/149 (70.5%)	105/105 (100%)
3	105/149 (70.5%)	105/105 (100%)
4	105/153 (68.6%)	105/105 (100%)
5	105/153 (68.6%)	105/105 (100%)
10	105/150 (70.0%)	105/105 (100%)
50	105/150 (70.0%)	105/105 (100%)
100	105/149 (70.5%)	105/105 (100%)

(a) Precision/Recall

Function	Num. insns
`fts-cycle.c:AD_compare`	30
`cp-hash.c:src_to_dest_compare`	30
`du.c:entry_compare`	30
`ls.c:dev_ino_compare`	30

(b) Hash collisions

Figure 5.7: a. The precision and recall rates for PBI for $n_{PBI} = 1, 2, \ldots, 5, 10, 50, 100$, the number of perturbation input-output pairs keeping $n_{TBI} = 10$. b. The largest group with hash collisions consisting of functions that compare between data structures. The last two functions, `entry_compare` and `dev_ino_compare`, were found to be copy-and-pasted code clones. The other false matches were due to insufficient code coverage.

Data set	n_{PBI}	Precision	Recall	F_2
`gccO3`	0	88/146 (60.3%)	88/90 (97.8%)	87.0
	1	87/133 (65.4%)	87/88 (98.9%)	89.7
	2	**87/128 (68.0%)**	**87/88 (98.9%)**	**90.7**
	5	87/132 (65.9%)	87/88 (98.9%)	89.9
	10	86/128 (67.2%)	86/88 (97.7%)	89.6
`gccOs`	0	91/151 (60.3%)	91/98 (92.9%)	83.8
	1	91/138 (65.9%)	91/98 (92.9%)	85.9
	2	**91/135 (67.4%)**	**91/98 (92.9%)**	**86.4**
	5	91/139 (65.5%)	91/98 (92.9%)	85.7
	10	91/139 (65.5%)	91/98 (92.9%)	85.7
`clangO2`	0	85/139 (61.2%)	85/86 (98.8%)	88.0
	1	80/111 (72.1%)	80/86 (93.0%)	87.9
	2	**83/114 (72.8%)**	**83/86 (96.5%)**	**90.6**
	5	84/122 (68.9%)	84/86 (97.7%)	90.2
	10	80/106 (75.5%)	80/86 (93.0%)	88.9
`clangO3`	0	80/134 (59.7%)	80/85 (94.1%)	84.4
	1	75/106 (70.8%)	75/85 (88.2%)	84.1
	2	79/115 (68.7%)	79/85 (92.9%)	86.8
	5	**81/121 (66.9%)**	**81/85 (95.3%)**	**87.8**
	10	76/104 (73.1%)	76/85 (89.4%)	85.6
Overall	0	344/567 (60.7%)	344/359 (95.8%)	85.9
	1	333/485 (68.7%)	333/357 (93.2%)	87.0
	2	**340/495 (68.7%)**	**340/357 (95.2%)**	**88.4**
	5	343/515 (66.6%)	343/357 (96.1%)	88.3
	10	333/477 (69.8%)	333/357 (93.3%)	87.4

Table 5.3: Precision and recall for *coreutils* compiled with `gcc -O3` (`gccO3`), `gcc -Os` (`gccOs`), `clang -O2` (`clangO2`) and `clang -O3` (`clangO3`), keeping $n_{TBI} = 10$. The reference set used was *coreutils* compiled with the default setting `gcc -O2`.

5.11 Discussion

5.11.1 Undefined behaviour

Since perturbing variables has the potential to produce undefined behaviour in code, the ways in which undefined behaviour occurred in practice were examined. Numerous instances of segmentation faults and three instances of an infinite loop were observed as a result of perturbation. The segmentation faults were caused by invalid memory dereferences, either because of an invalid pointer, or an invalid array index.

Two solutions were adopted to deal with this issue. The first was to instrument a simple check before each memory dereference to see if the address was a valid one. If this check failed, the function was made to immediately exit. The 9 most significant bits of `ESP` was used as the definition of the valid address space, and it was found that this worked for all but one case. For this last case, the array index slightly exceeded the array bounds whilst being updated in a for-loop. This was not caught by the pointer check and this perturbation site was removed.

In the second approach, it was observed that the array size was often passed as a parameter. Thus, to prevent the size from being too large, whenever an array was found to be one of the function parameters, the value of all integer parameters was capped to that of the array size. A more rigorous approach to deal with the problem of invalid memory dereferences is to use fat pointers [184, 185]. However, it was observed that, thus far, undefined behaviour was sufficiently mitigated by the above-mentioned simple techniques so as not to have it severely affect the detection rate.

The infinite loops were all caused by perturbations, in the form of increment statements `++i`, made to a decrementing loop counter, e.g.   `i`, causing non-termination of the loop. The solution was to manually remove the affected perturbation sites.

5.11.2 Indirect jumps and external code

An assumption that was made in this chapter was that the control-flow is statically known. In reality, there exists indirect jumps, for example due to virtual functions, as well as the use of function pointers whose targets are not statically known. One possible solution is to collect and perform analysis on execution traces instead of a static disassembly, relying on automated code coverage tools to fill in gaps in the control flow.

At the moment, what has been considered are internal functions, that is, functions that do not rely on external code. Analysing functions that, for example, make system calls is more involved and the approach used by KLEE [177] is to use the `uClibc` C library, or micro-Controller C library. Another approach used by Jiang and Su [176] is to treat the return value of a system call as another input to the function, eliminating the need to implement the call.

5.11.3 Function prototyping

As mentioned, the function prototypes were determined using the output of `gcc -aux-info`. However, approximating `void *` with `char *` caused segmentation faults in some cases, as they pointed to a data structure and had to be dealt with as such. The solution was to edit the function prototypes and replace `void *` with the appropriate `struct` pointer.

As a matter of practical consideration, three data types were found to be sufficient: `int`, `char[]` and `char **`. All `bool`, `char` and `short` variables could be approximated by an `int`, since they are passed as a 32-bit value on the stack; all `long long` 64-bit arguments could be approximated with two `int`s; arrays, both one and two dimensional, could be approximated by a large enough `char` array; finally all array of pointers could be approximated by the use of `char **` initialised to point to a pre-determined number of `char` arrays. In most cases, a `struct` could either be approximated with `char[]` or a `char **` of the appropriate size, with the exception of seven cases which had to be initialised as per its constituent member types to prevent segmentation faults. The overall implication of this is that three type categories: non-pointers, pointers and arrays of pointers, are sufficient for matching and indexing the majority of variables.

5.12 Related work

The work that is closely related to TBI/PBI is that of EqMiner [176], which also makes use of randomised input to find functionally equivalent code using input-output behaviour. EqMiner does not deal with intermediate state; instead, it extracts consecutive subsequences of statements, which allows it to match, for instance, fragments of bubblesort and selectionsort. Matching statement subsequences is a complementary technique to perturbation analysis, although perturbation analysis can match at the granularity of variables while that of subsequence matching is limited to matching between statements, which may involve more than one variable. BinHunt [175] performs semantic matching on basic blocks using symbolic execution and a theorem prover. Matching is done via pair-wise comparisons of all symbolic formulae generated. However, symbolic execution is a computationally expensive technique to apply on whole functions [175].

A related problem is software plagiarism and code clone detection. This has been an issue since at least the 1980s, when IBM brought lawsuits against firms that had cloned the PC-AT ROM based on 'software birthmarks'—the order in which registers were pushed and popped [186]. Several techniques have been proposed since, most notably program dependence graphs [164], directed acyclic graphs [187] and program expression graphs [165]. The commercial tool BinDiff [188, 150] does binary similarity analysis, first at the callgraph level, then at the procedure level by performing graph matching, then at the basic block level using instruction mnemonics. Another binary matching tool is Pierce and McFarling's BMAT [149], originally designed to propagate profile information from an old, well-documented build to a newer one for testing purposes.

6

Prototype recovery via inlined data source tracking

"Type information encapsulates much that distinguishes low level machine code from high level source code."

– Mike Van Emmerik [13]

As opposed to high level source code, type information is non-existent in machine code. Type recovery for machine code is thus an important problem in decompilation and in reverse engineering in general, as evident from the numerous papers that have been written on the subject [12, 26, 39, 13, 29, 189, 190, 36].

A sub-problem in this domain is the recovery of function prototypes. Prototype recovery is important for at least two reasons. Firstly, prototypes are important for interprocedural analysis [11]; secondly, prototypes are needed for determining valid input to a procedure. In our context, prototype recovery is needed to perform perturbation analysis, discussed in the previous chapter (Chapter 5), so that the appropriate number and type of arguments can be placed on the stack or in the registers.

Previous work in prototype recovery has focussed on static methods and on basic types such as integers and string constants [11]. However, static methods have to deal with at least two issues. Firstly, pointers in machine code induce aliasing problems, more so than in high-level source code, due in part to complex pointer expressions. This problem is further aggravated by the stack, which must first be logically split into local variables, register spills, parameters, caller and callee saves and compiler-generated storage such as return addresses and frame pointers. Secondly, variables with more than one live range can either be split into separate variables or united as one. This decision to split or to unite, which can affect the precision of the type assignment, is undecidable in general [13].

This chapter adopts an approach to prototype recovery that is based on dynamic analysis and differs from previous approaches in two fundamental ways. Firstly, it does not make the assumption that an existing set of test input is available. Indeed, dynamic analysis will not be meaningful without first inferring valid input to the procedure. Secondly, the approach exploits the fact that in practice function prototypes congregate into a finite and manageable number of categories, enabling a simpler type system to be used, yet still being able to recover complex parameter types such as recursive data structures. As a

coreutils			*linux*		
Freq.	Argslist	Percentage	Freq.	Argslist	Percentage
758	`N`	18.4%	83,845	`P`	31.3%
697	`P`	16.9%	38,438	`PP`	14.4%
372	`PP`	9.0%	31,019	`-`	11.6%
351	`-`	8.5%	27,124	`PN`	10.1%
272	`PN`	6.6%	12,469	`PPP`	4.7%
267	`NN`	6.5%	9,356	`PNN`	3.5%
141	`PPN`	3.4%	8,977	`PPN`	3.4%
132	`NP`	3.2%	8,774	`N`	3.3%
107	`NQ`	2.6%	4,959	`PNP`	1.9%
88	`PNN`	2.1%	3,598	`PPPN`	1.3%

Table 6.1: The frequency and percentage of the most common function arguments (argslists) in *coreutils* and *linux* (`N` refers to a non-pointer, `P` represents a pointer, `Q` represents a pointer to pointers).

side note, an additional benefit of performing dynamic analysis is it naturally fits with perturbation analysis.

This chapter makes the following contributions:

Of the function prototypes in the GNU coreutils tool suite 6.10 and the Linux kernel 3.0.1 surveyed, 99.0% had 6 parameters or fewer. Pointers to user-defined data structures, i.e. `struct X *`, was the most common parameter type. The majority of data structures was found to be either consisting of all non-pointers or all pointers, allowing them to be approximated by `char *` and `char **` types respectively (Section 6.1).

The algorithm for prototype recovery is inspired by software exploit detection and software testing. Inlined data source tracking (Section 6.6), the primary type inference mechanism, was inspired by the principle of conformant execution; probabilistic branch negation is an additional technique used to improve code coverage (Section 6.7).

Unlike prior prototype recovery algorithms, the algorithm is able to recover complex types, such as recursive data structures.

The algorithm is able to correctly recover 84% of function prototypes from *coreutils*, and 62% of function prototypes from *glibc*, out-performing the current state-of-the-art prototype recovery algorithm. The worse case timing is 0.4 seconds per function for *coreutils* and 1.65 seconds for *glibc* (Section 6.11).

6.1 Survey of prototypes in *coreutils* and *linux*

Two data sets were surveyed: 4,115 functions from the *coreutils* 6.10 tool suite, and 267,463 functions from the *linux* kernel version 3.0.1.

A prototype can be divided into the list of arguments, the *argslist*, and the return type. By grouping all data types into three categories—non-pointers (N), pointers (P) and pointer to pointers (Q)—the first task was to count the frequency of argslist categories expressed in terms of N, P and Q. The prototypes were obtained using the gcc compiler option --aux-info. The most commonly occurring argslist categories are listed in Table 6.1. There were differences in the order of the most common arglists, although the single pointer (P), double pointer (PP) and void argument categories (-) were among the most common in both data sets. What was interesting was that the frequency order: P, PP, -, was the same for both sets of functions.

There was, however, more consistency in the number of parameters per function. Approximately 35% of functions had one parameter; about 28% had two; 15% had three; and about 10% had no function parameters (Figure 6.1). The function with the most parameters was the cq_enet_rq_desc_dec (Cisco ethernet receive descriptor decode) function from *linux* with 29. Function prototypes with six or fewer parameters accounted for 99.0% of all prototypes for both data sets.

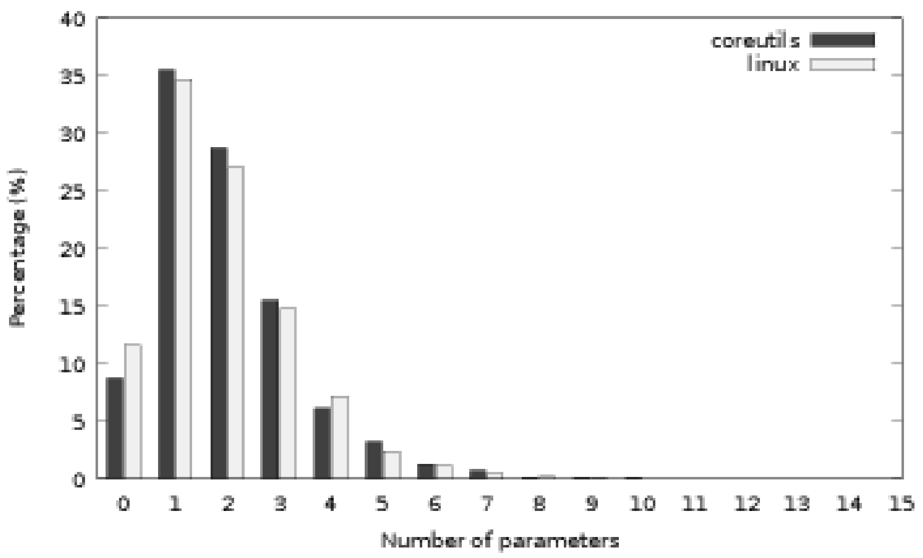


Figure 6.1: Number of parameters for functions in *coreutils* and *linux*.

In terms of individual parameter types (500,000 in total), there were 354,248 occurrences of a pointer type, or 70.0%, among all parameters; 149,966, or 29.7%, were non-pointers; pointer to pointers numbered 1,565, or 0.3% of all parameters. Among the pointer types, pointers to user-defined data structures, or struct types, were the most common (55.2%), followed by pointers to a basic type, for example char * or int * (13.9%), then by union types (0.3%) and finally function pointers (0.1%).

Next, a closer examination of the struct types in use was performed, since struct pointers were the most popular parameter type overall. Pointer types were similarly mapped to P, non-pointers to N and pointer to pointers to Q. There were a total of 39 and 11,256 different struct types in *coreutils* and *linux* respectively. The ten most popular struct layouts are listed in Table 6.2. Interestingly, the two most frequently occurring structs consisted of two and six non-pointers, accounting for 27% and 32% of all structs in coreutils and linux respectively. Aside from a few exceptions, the most frequently occurring data structures consisted of either all non-pointers or all pointers, accounting for more than 50% of all structs.

coreutils			*linux*		
Freq.	Members	Percentage	Freq.	Members	Percentage
6	`NN`	15.4%	2,333	`NNNNNN`	20.7%
5	`NNNNNN`	12.8%	1,241	`NN`	11.0%
3	`PPPPPP`	7.7%	934	`NNN`	8.3%
3	`PPP`	7.7%	746	`NNNN`	6.6%
2	`PPPPPN`	5.1%	664	`N`	5.9%
2	`PPPPP`	5.1%	541	`PPPPPP`	4.8%
2	`NP`	5.1%	480	`NNNNN`	4.3%
2	`NNP`	5.1%	294	`PP`	2.6%
2	`NNNNN`	5.1%	244	`P`	2.2%
1	`PPPP`	2.6%	191	`PNNNNN`	1.7%

Table 6.2: Most common constituent member types of user-defined `struct` types found in *coreutils* and *linux*.

The implication of Table 6.2 is that from the perspective of performing type recovery, the majority of data structures can be approximated with either a large representative `char` array, for structures of the form `NNNN...`, or a pointer to pointers, for structures of the form `PPPP....` This simplifies the type recovery process significantly since we need only consider three type categories: `P`, `N` and `Q`.

6.2 Algorithm design

We observe that it is rarely intentional for a function to use memory that has not been uninitialised by its caller or access memory addresses that are out-of-bounds. Hence, one approach is to use these two criteria, uninitialised memory and invalid pointers, to validate candidate prototypes: invalid solutions should fail to meet one or both conditions; conversely, valid solutions should satisfy both conditions. This is analogous to a valid type assignment satisfying all of its type constraints.

This chapter makes the following assumptions.

1. It is assumed that f does not contain uninitialised memory errors.
2. It is assumed that complete control-flow information is available. Control-flow reconstruction is considered to be a separate problem from prototype recovery.
3. It is assumed that standard x86 calling conventions, such as stack-based and fastcall conventions, are used.

The proposed algorithm is based on *conformant execution*, which is described now.

6.3 Conformant execution for type recovery

A concept recently coined by Jacobson et al., an execution trace of a program is said to be *conformant* to a *policy*, which is a set of requirements, if it satisfies that policy for all its program states [191]. The aim of enforcing a conformant execution policy is to stop an attacker from exploiting a software vulnerability in a program in order to execute his or her code. For example, in the context of preventing code injection-based exploits such as buffer overflow exploits, an appropriate policy is to require that the instruction pointer cannot point to memory that has both the write and execute bits set. This is also known as the $W \quad X$ data execution prevention policy [192]. As a second example, in the context of preventing code-reuse attacks, such as return-oriented programming in which an exploit populates the call stack with a list of addresses in order to execute several short instruction sequences that end with a `retn` instruction, an appropriate policy is to check the validity of all control-flow source-target pairs, also known as maintaining control-flow integrity [193].

While conformant execution is useful for detecting unintended code execution, in this chapter it is adapted for the purposes of type recovery. The main premise is that a valid prototype, when translated to an execution trace, should conform to certain validity policies, for example those defined in Sections 6.4 and 6.5.

The goal is to infer the prototype of a function, which involves inferring the argslist and the return type. An argslist defines an *input state*, and the two terms are used interchangeably. A program state, s, is represented by the tuple

$$s = \quad a_1, \ldots, a_m, x_1, \ldots, x_n, v$$

where a_i are the input parameters, x_j are the program variables and v is the output variable. At the input state, s_0, only a_i are defined. At the final state, s_f, v is defined. Given an input state s_0 and a policy F, s_0 is valid if and only if subsequent states conform to F, i.e. $F(s_1) \quad F(s_2) \quad \ldots \quad F(s_f)$ is satisfied. Policy F determines the precision of the analysis: if F contains too many requirements, or is too restrictive, then no valid s_0 may be found; conversely, if F contains too few requirements, or is too permissive, then too many s_0 are valid solutions. Our policy for prototype recovery is made up of two components, namely memory validity, F_{mem}, and address validity, F_{addr}.

6.4 Memory validity, F_{mem}

The goal of the memory validity policy, F_{mem}, is to ensure that no uninitialised memory is used by the function. Although the use of uninitialised memory may be an error in the function, this is rarely intended and the assumption made is that a function given valid input does not contain such errors.

Consider the following caller-callee code snippet which follows a typical stack-based calling convention. The dereferencing of pointer `EBP + 8` at line `g3` is of interest, as it may cause an uninitialised value on the stack to be used. Assuming the stack is empty at instruction `f1`, there are three items on the stack by the time execution reaches instruction `g3`: the constant 3, the return address `f3` and the saved `EBP` from `g1`. Therefore, `EBP + 8` logically points to a value placed by the caller function f, in this case 3.

```
f:
...
f1:  push  3
f2:  call  g
f3:  ...

g:
g1:  push  EBP
g2:  mov   ESP, EBP
g3:  mov   EDX, [EBP + 8]
...
```

However, given only instructions `g1`, `g2` and `g3`, what is inferred is that the four bytes located at `EBP + 8` are in use in g, but their type is not known.

To implement F_{mem}, the heuristic that the contents of uninitialised memory remains the same before and after the arguments are written by the caller is used. Therefore, instrumentation is inserted before the first instruction so that the stack and CPU registers (excluding the stack and frame pointer) are populated with a predefined known magic value, say `MAGIC` (lines `g0i` and `g0ii` in Figure 6.2). The `initInputState` function simulates the caller function, which populates the input parameters (line `g0`). For instrumentation inserted *before* an instruction, we use the Roman numerals `i, ii, ...`. For instrumentation inserted *after* an instruction, we use the suffixes `a, b, ...`. We use the `__C ...` notation to indicate inlined C, analogous to the `__ asm ...` notation for inlined assembly.

Next, before every pointer dereference or register use an assertion is inserted to test if the memory pointed to or the register contains the magic value. Where the register read is a sub-register, e.g. `AL`, the test is made to the whole register, i.e. `EAX`. Suppose the prototype, x, was the empty argslist, `-`. Upon execution, the assertion fails at line `g3i` since the address location `EBP + 8` is uninitialised, F_{mem} is violated and thus x is not valid. On the other hand, if x was the argslist `N` consisting of a single integer, then the four bytes at location `EBP + 8` is populated and the assertion evaluates to `true`. Note that the argslist `NN` will similarly be valid since it also produces a conformant execution trace. Here the principle of *Occam's razor* is used and `N` is chosen over `NN` since it is the simpler explanation.

There is, however, the issue of false positives since some registers are used as temporary local variables and have their contents read to be stored on the stack at the beginning of the function. The process of reading and storing these registers will be flagged as violating F_{mem}. Instructions such as `xor EAX, EAX` also generate false positives, since although syntactically register `EAX` is read, semantically its value is over-written. To deal with the first case, only the fastcall registers, `EAX`, `ECX` and `EDX` are pre-populated with `MAGIC`. To deal with the second case, these instructions are not instrumented.

6.5 Address validity, F_{addr}

As mentioned, it is assumed that a function, given valid input, should not access out-of-bounds memory. The code snippet in Figure 6.3 gives an example of a potential out-of-bounds memory read. In instruction `g4`, the memory address being dereferenced is `EDX`.

```
g:
           __C
g0i:         int stack[max];
g0ii:        for(i=0;i<max;i++)   stack[i] = MAGIC;
g0:          initInputState();

g1:        push  EBP
g2:        mov   ESP, EBP
           __C
g3i:         assert( *(int *)(EBP + 8) != MAGIC, "F_mem violated" );

g3:        mov   EDX, [EBP + 8]
...
```

Figure 6.2: An example of instrumenting a function to implement policy F_{mem}.

```
g:
           __C
g0i:         int stack[max];
g0:          initInputState();

g1:        push  EBP
g2:        mov   ESP, EBP
g3:        mov   EDX, [EBP + 8]
           __C
g4i:         assert( EDX >= &stack[0] &&
              EDX <= &stack[max], "F_addr violated" );

g4:        mov   EAX, [EDX]
...
```

Figure 6.3: An example of checking address validity, which constitutes policy F_{addr}.

Policy F_{addr} constrains memory addresses to be within the bounds of the stack, and this is asserted prior to executing g4 in instruction g4i. If this assertion fails for argslist x, F_{addr} is violated and x is not a valid solution. In practice, this policy will not infer any valid solutions for some functions since global variables lie outside of the local stack frame. This issue is addressed by inlined data source tracking (Section 6.6).

The implementation of both F_{mem} and F_{addr} enforces the policy that uninitialised memory is not read and out-of-bounds memory access is not permitted.

However, there are two aspects to finding a valid x: firstly, the data structures involved need to be inferred; secondly, there is also a need to infer the appropriate values with which to populate those data structures. While simple, the downside of using only F_{mem} and F_{addr} is that many executions of f are potentially required to find a valid x. In the next section, a technique, called *inlined data source tracking* (IDST), is proposed. IDST addresses these two issues and quickly finds candidate data structures from only one or a few executions, accelerating the process of reconstructing a valid x.

6.6 Inlined data source tracking

It is observed that traversing a data structure consists of pointer dereferences and pointer arithmetic. The addressing modes used in pointer dereferences tend to be one of the following four types: base-only, e.g. `[EAX]`, base-immediate, e.g. `[EAX + 4]`, base-indexed, e.g. `[EAX + EDX]`, and base-scaled, e.g. `[EAX + 4 * EDX]`. Pointer arithmetic tends to be simple addition involving small constants. Pointer subtraction is rare in practice and is currently not dealt with, but it is not difficult to implement.

The key idea of inlined data source tracking (IDST) is to exploit the fact that the base pointer of a data structure is likely to be constant in its upper, or most significant, bits, with its lower bits updated to point to the various data structure components. Thus a data structure can be tracked by storing its source information in the upper bits of the base pointer, and only rewriting the pointer with a proper address immediately prior to an dereference.

The requirements for the tracking scheme were the following.

1. The *data sources* to be tracked include all input parameters and global variables.
2. The number of successive pointer dereferences made using the same base pointer needs to be tracked, thus allowing the separation of numbers, arrays, pointer-to-pointers and recursive data structures.
3. The scheme has to allow pointer arithmetic, particularly the addition of small offsets to the base pointer.
4. The scheme should not track variables that are not data structures. Thus tracking is to cease when the variable is not used as a pointer, for example when it is used in multiplication, division or logical operations.

Two schemes were considered: an inlined scheme and a shadow tracking scheme. The latter scheme was envisaged to use shadow registers and shadow memory. The advantage of such a scheme is the strict non-interference between tracking data and the actual computation data. However, the main disadvantage of this scheme is the complexity of implementing the shadow memory and the shadow computation to be performed for all the different instruction types in use.

Compared to a shadow tracking scheme, inline tracking is more expedient. The potential downside, however, is the possible interference between the tracking data and the actual computation data which could cause incorrect tracking information to be reported. To deal with requirement 3, all tracking information is encoded in the upper most significant 16 bits, leaving the lower 16 bits for pointer arithmetic. To deal with requirement 4, 8 bits are reserved for use as a “tamper” seal which would be destroyed should the variable be used in multiplication, division or logical operations. This 8-bit magic number could be placed altogether, or interspersed with tracking data. The choice was made to place the magic number the beginning and at the end of the upper 16 bits.

The tracking scheme consisted of the following components (Figure 6.4).

An 8-bit magic value, `MAGIC`, split into two groups, which indicates that this variable is still being tracked.

MAGIC (4 bits)	Dereference counter (4 bits)	Source identifier (4 bits)	MAGIC (4 bits)	Offset (16 bits)

Figure 6.4: The encoding scheme for inlined data source tracking.

A 4-bit dereference counter, `deref_count` which tracks the number of successive dereferences made with a pointer.

A 4-bit data source identifier, `source_ID`.

A 16-bit base pointer offset value which allows pointer arithmetic to be performed without interfering with the tracking. Offsets are assumed to be 2^{16} or smaller.

To implement this scheme, the function is instrumented at four different sets of locations: at the start of the function, before every register use, before every pointer dereference and after every pointer dereference. In addition, a segment of the stack is set aside as scratch space, `scratch[smax]`, and a saved slot is reserved for saving and restoring base address pointers.

1. At the start of the function, all data sources are initialised, that is, the 8-bit magic value is set, the dereference counter is set to 0, the source identifier is set to the unique value assigned to that source.
2. Before every register use, a check is made to see if the 8-bit magic value is present. If so, the entire 32-bit value of the register is recorded.
3. Before every pointer dereference, the base address register is checked for the 8-bit magic value. If it is present, its value is copied to the saved slot, its dereference counter is incremented by one, the 32-bit value is recorded, its 32-bit value is written to every element in `scratch`, and lastly the base address is assigned the value of `&scratch`. Since the base address now points to `&scratch`, future dereferences will use the new encoded value.
4. After the pointer dereference, the base address register is restored with the value in the saved slot.

In the final implemented scheme, instead of having both F_{mem} and F_{addr} policies, the F_{mem} policy is replaced by inlined tracking, while the F_{addr} policy is retained.

An example of inlined data source tracking is given in Figure 6.5. The output log indicates that register `EAX` is an input parameter from instruction `g3` and is a pointer since its dereference counter is one. Register `EDX` is not an input parameter, since it is not read. Since `EDX` is overwritten in instruction `g4v`, it is no longer tracked after instruction `g4`, but future reads from the `scratch` array will continue to track the data structure.

One can leverage this scheme to also infer recursive data structures, such as linked lists and trees. We say a data structure A is recursive if A contains a member that is a pointer

```
g:
          __C
g0i:        int scratch[smax];
g0ii:       EAX = 0xA00A0000; /* source_ID = 0 */
g0iii:      EDX = 0xA01A0000; /* source_ID = 1 */
g0:         initInputState();

g1:       push  EBP
g2:       mov   ESP, EBP
          __C
g3i:        if( contains_magic( EAX ) )
g3ii:        log( "%x n", EAX );

g3:       mov   EDX, EAX
          __C
g4i:        if( contains_magic( EDX ) )
g4ii:        EDX = increment_deref( EDX );
g4iii:       log( "%x n", EDX );
g4iv:        for(i=0;i<smax;i++)  scratch[i] = EDX;
g4v:         EDX = (int)&scratch;
g4vi:

g4:       mov   EDX, [EDX]
...
          Expected output:
l0:       A00A0000
l1:       A10A0000
```

Figure 6.5: An example of inlined data source tracking and the expected output after execution. The `MAGIC` bits, which occupy bits 16 to 19 and 28 to 31 are `0xA00A0000`. Line `l0` indicates that the `EAX` register (`source_ID` = 0) is an input parameter and line `l1` indicates that `EAX` is a pointer (with `deref_count` = 1), while `EDX` (`source_ID` = 1) is unused.

to A. In practice, it is rare for non-recursive data structures to have 2^4 1 or more nested levels of pointer indirection. Thus, if the dereference counter for a structure X reaches 2^4 1, it is likely due to the iterative and recursive procedures required for enumerating a recursive data structure, and X is assigned the recursive type `R`. The corresponding data source is subsequently assigned the `NULL` value so that it is no longer tracked.

The return variable type is inferred using the heuristic that the return variable, v, is stored in `EAX`.

If v is still tracked, i.e. it contains a valid `MAGIC` value, the type of a parameter or return variable is derived from the value of `deref_count` in the following manner.

$$v = \begin{cases} \texttt{N}, & \texttt{deref\_count} = 0 \\ \texttt{P}, & \texttt{deref\_count} = 1 \\ \texttt{Q}, & \texttt{deref\_count} \quad 2 \quad \texttt{deref\_count} < 2^4 \quad 1 \\ \texttt{R}, & \textit{otherwise} \end{cases}$$

Otherwise, v is assigned type N.

6.7 Probabilistic branch negation

One shortcoming of using a dynamic approach is the inability to reach all code. Two techniques have been proposed in the past to increase code coverage, which can be described as input data manipulation, also known as automated test-case generation [194, 177], and control-flow manipulation, used in tools such as Flayer [195]. The former approach was not considered as input data was used to store tracking information. On the other hand, the instruction semantics could be modified.

The technique adopted by Drewry and Ormandy is to manually convert one or a few conditional branches to unconditional ones [195]. A similar approach, called *probabilistic branch negation*, is proposed here. However, instead of one or a few conditional branches, all conditional branches are modified and we define the probability $P_b \quad [0, 1]$ that the branch is negated. For example, let

$$B : \texttt{JNZ} \quad \texttt{I;}$$

which is equivalent to

$$B : \texttt{if (!ZF) goto} \quad \texttt{I;}$$

in C, where ZF is the zero flag and I is the branch target. The negation of B is

$$B : \texttt{if (ZF) goto} \quad \texttt{I;}$$

Given a random number $r \quad [0, 1]$, the branch is negated with probability $P_b \quad [0, 1]$, i.e.

$$\texttt{if (r} \quad \texttt{P}_\texttt{b}\texttt{) B else} \quad \texttt{B.}$$

If $P_b = 0$, the instruction is unmodified from the original. If $P_b = 1$, the branch is always negated.

6.8 Type system

The six types in the type system are: , N, P, Q, R and . The partial order is given by

$$:> \texttt{N} :> \texttt{P} :> \texttt{Q} :> \texttt{R} :>$$

C type	Maps to
`bool, char, short, int`	`N`
`long long int`	`NN`
`char *,...,int *`	`P`
`void *`	`P`
`char[],...,int[]`	`P`
`char[][],...,int[][]`	`P`
`char **,...,int **`	`Q`
`void **`	`Q`
Data structure pointers whose largest type member = `N`	`P`
Data structure pointers whose largest type member <: `N`	`Q`
Recursive data structures	`R`

Table 6.3: Mapping of C types to the type system.

where :> is the subtype relation. The maximal element is "type not yet inferred"; the minimal element represents "type conflict", but it is unused in practice; type `N` represents "either a number or pointer-type"; type `P`, or "pointer-type", and is a subtype of `N`; type `Q` is a subtype of `P`, for example a pointer to a structure containing at least one pointer; finally the recursive data structure `R` is a subtype of `Q`.

The mapping of C types to this type system is as follows. Basic C types, such as `bool` and `char`, map to `N`. Arrays, both one and two dimensional, map to `P`. Pointer to pointers, such as `char **`, map to `Q`. For a data structure pointer `struct A *`, the member with the largest value in this lattice is used. If that member is `N`, then `struct A *` maps to `P`. Otherwise, if that member is `P` or higher, then `struct A *` maps to `Q`. For example, given a data structure `telephone` that has two members, `char *name` and `int number`, and a function that has the prototype `int getTelephoneNumber( struct *telephone )`, the first argument for `getTelephoneNumber` will be mapped to `Q` according to this mapping, since the parameter is a pointer to a `telephone` structure and the largest element of `telephone` is a pointer. The mapping of C types to this type system is summarised in Table 6.3.

We define the meet operator, as

$$x \quad y = \begin{cases} x, & if x <: y \\ y, & otherwise \end{cases}$$

The operator is used when combining solutions from multiple runs of the probabilistic branch negation algorithm.

6.9 Distance metric

The goal is to infer a type that is close to the actual type. For the purposes of evaluation, the distance metric is defined to evaluate the performance of the recovery.

Let the inferred prototype be $x = \quad v, x_0, x_1, \ldots$, where v is the return type and $x_0, x_1, \ldots$ are the parameter types. Let the actual prototype be $y = \quad u, y_0, y_1, \ldots$ and the number of parameters in a prototype p be denoted by $\quad p \quad$. Let D_l the lattice distance between element

x_i and y_i, be equal to their relative positions on the lattice. For example, $D_l(\texttt{N},\texttt{P}) = 1$. Let the maximum distance between two elements on the lattice be $D_{l\,\max} = D_l(\ \ ,\texttt{R}) = 4$. If x contains a different number of parameters than y, then $D_{l\,\max}$ is added to the distance for every parameter incorrectly missed or added. Thus, the distance is defined as

$$D(x,y) = D_{l\,\max} \quad abs\,(\quad x \qquad y\ \ ) \left(\sum_i D_l(x_i,y_i) \right) + D_l(v,u)$$

where $i = \min(\ \ x\ \ ,\ \ y\ \ )$.

For example, if $x = \ \ \texttt{P}\ \ $ and $y = \ \ \texttt{NN}\ \ $, then their distance is given by

$$\begin{aligned} D(\ \ \texttt{P}\ \ ,\ \ \texttt{NN}\ \ ) &= 4 \quad abs\,(1 \quad 2) + D_l(\texttt{P},\texttt{N}) \\ &= 4 + 1 \\ &= 5 \end{aligned}$$

6.10 Implementation

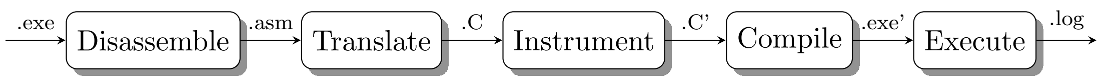


Figure 6.6: Overview of the IDST work flow.

Inlined data source tracking is implemented in five phases—disassembly, translation, instrumentation, compilation and execution (Figure 6.6). Disassembly and translation are performed using the Dyninst binary instrumentation framework. The translated program comprises two parts: a C file containing the assembly instructions written as macros, e.g. `MOV32( EAX, 0 )`, and a header file containing the macro definitions, e.g. `#define MOV32( dst, src ) dst = src;`. More details of the assembly-to-C translator, which consists of 1,546 lines of C++, can be found in Section 5.2. Compilation is performed using `clang`. Probabilistic branch negation was implemented as a second header file, with the conditional branch instructions modified to include the P_b variable and negated branch instructions. The following data sources were tracked: Fastcall CPU registers `EAX`, `ECX`, `EDX`, stack locations `[ESP + N]`, for $\texttt{N} = \ \ 4, 8, 12, 16, 20, 24\ \ $ and global variables.

6.11 Evaluation

The prototype recovery technique was evaluated on two data sets: 99 functions from coreutils 6.14, *coreutils*, and 343 functions from GNU C library 2.17, *glibc*. Both sets of functions were compiled with `gcc -O2`. Function prototypes were obtained using gcc -aux-info. All experiments were conducted on an Intel Core i5 2.5 GHz with RAM size of 4 GB. The results obtained were compared with the current state of the art algorithm used in the SecondWrite binary rewriting tool [46].

The pertinent research questions are the following.

How accurate is basic inlined data source tracking (IDST) in inferring function prototypes?

How effective is probabilistic branch negation (BN) in improving the performance of IDST?

What is the run-time cost of performing IDST?

6.11.1 Basic inlined data source tracking

In the first set of experiments, the accuracy of basic inlined tracking algorithm as per Section 6.6 was evaluated using the distance metric described in Section 6.9. The results for the `coreutils` and `glibc` data sets are shown in Figure 6.7.

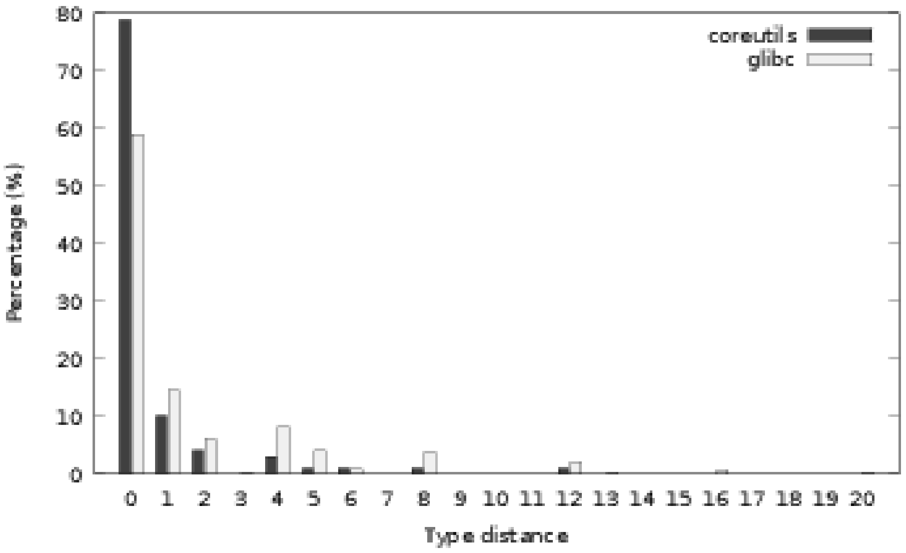


Figure 6.7: Type distance for functions in *coreutils* and *glibc* using basic inlined data source tracking with F_{addr}.

The number of prototypes correctly inferred, i.e. having a type distance of zero, was 78 out of 99 (78.8%) for `coreutils` and 202 out of 343 (58.9%) for `glibc`. The number of prototypes having a distance of at most four, equal to the distance of $D_{l\,\max}$, was 95 out of 99 (96.0%) for `coreutils` and 302 out of 343 (88.0%) for `glibc`.

On manual analysis, one of the reasons for the incorrect types inferred was due to poor code coverage. For example, string manipulation functions had `switch` constructs that only dealt with ASCII values, parameters that were incorrect buffer lengths and so on. Code coverage was addressed in the next set of experiments.

6.11.2 Adding probabilistic branch negation

Two schemes were considered: A purely probabilistic branch negation scheme (BN), and a hybrid scheme that combined the basic approach with probabilistic branch negation (Basic + BN). For the BN scheme, the function was executed eight times, and the overall solution was the over all eight solutions. For the Basic + BN scheme, the overall

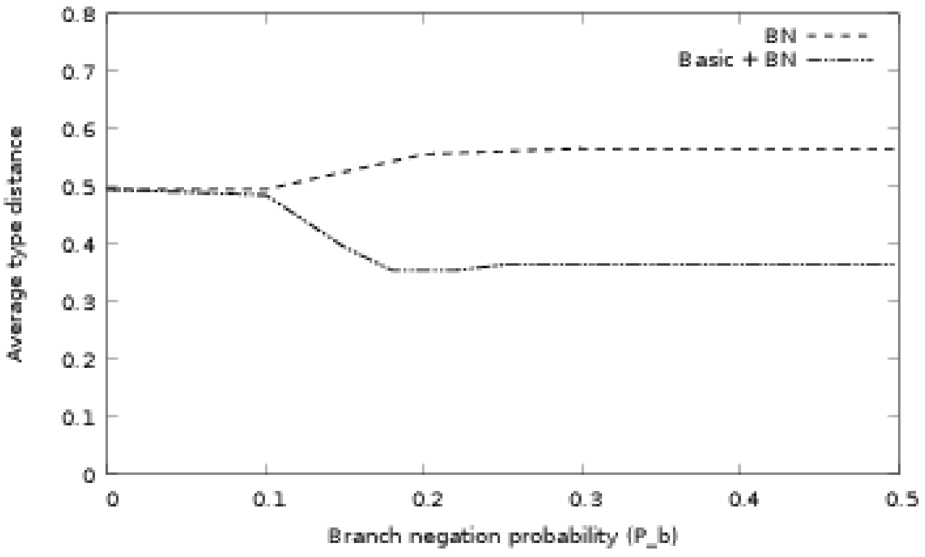


Figure 6.8: Branch negation probability, P_b versus average type distance for functions in *coreutils*, for BN only and Basic + BN.

solution was the of the basic solution and the eight BN solutions. Values of P_b from 0 to 0.5 were tested. When $P_b = 0$, there was no branch negation, and the BN scheme was equivalent to the Basic scheme. When $P_b = 0.5$, there was a 50% probability of the branch being negated. The results obtained for *coreutils* are shown in Figure 6.8.

At $P_b = 0$, there was no difference between the BN and Basic schemes, which was to be expected. However, for $P_b > 0$, the BN scheme had a performance that was worse than the Basic scheme, with an average type distance of 0.56 at $P_b = 0.2$, which increased to 0.57 for $P_b > 0.2$. This was surprising as it appeared that the code coverage seemed to be poorer compared to the Basic scheme. One explanation is that the paths executed when using BN were different from the ones executed otherwise.

When combined with the Basic scheme, Basic + BN inferred prototypes that had an average type distance of 0.354 for *coreutils*, peaking at $P_b = 0.2$ and identifying 84% of prototypes. This confirmed our hypothesis that the paths executed were mutually exclusive under the two schemes. Thus Basic + BN was an improvement over the basic algorithm, which only managed an average type distance of 0.495, and was able to identify 78.8% of prototypes.

Similar results were obtained for *glibc*. The Basic + BN scheme outperformed both BN and Basic schemes, with its performance peaking at $P_b = 0.2$ and inferred prototypes having an average type distance of 1.33. A total of 62.4% of prototypes were inferred correctly. In contrast, the basic scheme only managed an average type distance of 1.52 and inferred 58.9% of prototypes correctly (Figure 6.9).

Comparing our results with that of SecondWrite, the false positive rate reported by El-Wazeer et al. was 0.2 for arguments and 0.44 for return variables [46]. The false positive rate was defined to be the difference between the number of arguments or return variables inferred and the actual number, i.e. the $abs(\ x \quad y\ )$ component of the distance metric defined in Section 6.9. In comparison, the false positive rate for the Basic + BN method was 0.175 for *coreutils* and *glibc* combined.

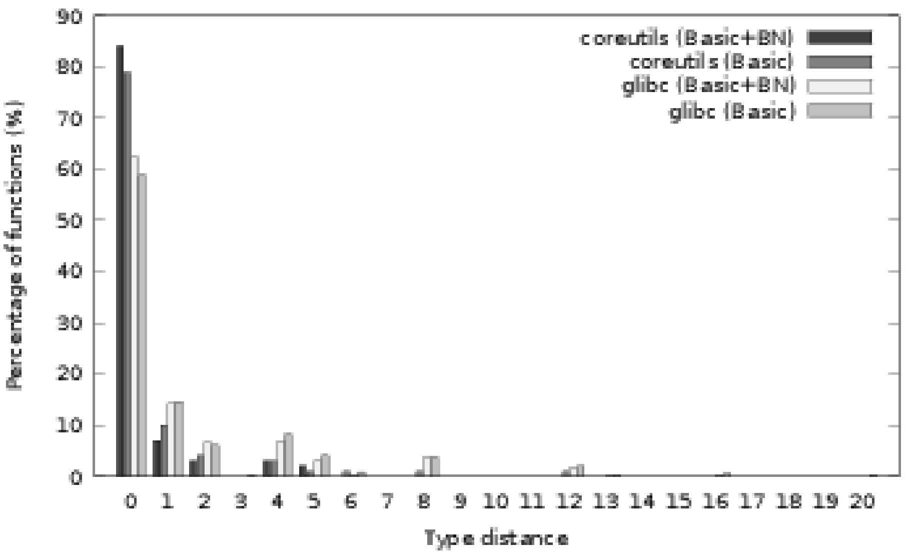


Figure 6.9: A comparison of the Basic approach versus Basic + BN.

	coreutils Basic		*coreutils* Basic + BN		*glibc* Basic + BN	
Phase	Average (s)	Worst (s)	Average (s)	Worst (s)	Average (s)	Worst (s)
Disassembly Translation Instrumentation	16.4	16.6	17.3	17.3	470.7	483.0
Compilation	10.2	10.7	20.4	20.4	50.7	51.8
Execution	0.56	0.69	3.0	3.1	30.2	31.1
Total for all functions	27.2	28.0	40.7	40.8	551.5	565.9
Average per function	0.275	0.283	0.411	0.412	1.61	1.65

Table 6.4: Average and worst case timing for the Basic and Basic + BN algorithms.

6.11.3 Timing

The timing of the Basic + BN algorithm can be split into three phases—disassembly to instrumentation, compilation and execution of the instrumented binary.

The average and worst case timings for three configurations are listed in Table 6.4. Comparing Basic and Basic + BN, the disassembly phases did not differ significantly. However, compilation took twice as long in Basic + BN since the instrumented C program is compiled once with P_b defined and once without. Overall, Basic was about 30% faster than Basic + BN, but about 12–28% less accurate in terms of average type distance.

The per function timing for *coreutils* was a quarter that for *glibc*, but both were well under two seconds. The function that took the longest time to analyse was `sha256_process_-block`, which contained 3,366 instructions and took 8.2 seconds of otherwise reasonable processing time.

6.12 Discussion

While the aim of IDST is to recover the function prototype, it is the first step in the larger goal of recovering the types of other variables. One approach to extend the current

```
static const char *file_name;

/* Set the file name to be reported in the event an error is detected
   by close_stdout. */
void
close_stdout_set_file_name (const char *file)
{
  file_name = file;
}
```

Figure 6.10: IDST was unable to correctly infer `file` as a pointer as it was never used as one.

implementation to perform type inference is to use IDST to obtain valid input. The rest of the variables can then be recovered via techniques such as REWARDS [189] or Howard [190].

IDST failed to find the right type in a few cases. IDST was unable to infer parameters that were never read, in which case IDST assumed fewer parameters than were actually present. For example, a parameter that was read but never used as a pointer was assigned the type `N`. Similarly, a pointer-to-pointers that was only used as a pointer, was assigned the type `P`. For instance, the function `close_stdout_set_file_name` had a pointer type `P` as its parameter, but IDST assigned `N` due to the fact the parameter `char *file` was read and assigned to the global variable `char *file_name` but was never dereferenced (Figure 6.10).

A related issue was the approximation of data structures to its largest type. Functions with parameters that were data structures did not always use all of its members, in particular its largest type. In such cases IDST inferred a type that differed from its mapped type.

Another case where IDST had difficulty involved the `void *` C type. A `void *` is type ambiguous in that it "can be converted to or from a pointer to any incomplete or object type" [196]. An example is the function `insque` that inserts an element into a doubly-linked list. It has two parameters, declared `void *` type but cast to `struct qelem *`, which is a recursive data structure. Another example is `obstack_allocated_p` which determines whether an object has been allocated from an object stack data structure. The object parameter is declared as a `void *` as it can be of any object type. IDST inferred the object as type `N` since it is merely used as an address and not dereferenced.

Thus far only leaf functions have been considered. Extending this technique to other functions requires dealing with API and system calls. For POSIX system calls, there is `uClibc`, which is a small C library optimised for embedded platforms. However, this is not possible for API calls that are not open sourced. Another option is to perform execution environment modelling [197] which uses program synthesis methods to construct a model given a specification for an API call.

6.13 Related work

Previous work on type recovery in machine code can be categorised into four main groups—the constraint-based approach [12, 29, 36], the data-flow analysis approach [13], the REWARDS information-flow approach [189] and the Howard method, which uses dynamic memory usage modelling [190]. The first two approaches are primarily static whilst the latter two are dynamic in nature. The exception is TIE [36], which can be used statically or dynamically. The approach proposed here is based on the principle of conformant execution, which hitherto has not been used in the context of type recovery. The difference between IDST and that of REWARDS, Howard and TIE is that IDST does not assume to receive valid input, whereas they do. The goal is thus also different—IDST seeks to infer the function prototype, although like Howard and TIE the aim is to recover higher level data structures.

Inferring function prototypes is a sub-problem to type recovery. The method of Cifuentes is based on liveness analysis and reaching definitions. Parameter types considered included basic types, such as 8-, 16- and 32-bit integers, and string constants where API call parameter type information was available [11]. Zhang et al. observed that a parameter is defined by an ancestor function and used by its descendent. Conversely, return variables are defined by a descendent function and used by its ancestor. Thus identifying parameters and return variables is framed as identifying data dependence edges that cross function boundaries. Arrays and structures are not supported [198]. BCR is a binary code reuse tool that identifies parameters and return variables from multiple execution traces using a combination of instruction idiom detection, taint analysis and symbolic execution. The availability of one valid input is assumed; types up to a pointer to a single-level data structure are supported [63]. Anand et al. framed the problem of recovering function parameters as one of determining value ranges of address expressions, using value-set analysis (VSA). Where VSA is too imprecise, run-time checks are instrumented in the binary. However, it is unclear how valid input is obtained to enable run-time analysis [45, 46]. Unlike prior prototype recovery approaches, IDST is able to recover recursive data structures and no valid input is assumed.

Conformant execution was recently coined by Jacobson et al. as a means to stop code reuse attacks [191]. However, the same term can be generalised to other memory corruption error detection and prevention mechanisms. Memory corruption protection techniques may be grouped according to what they protect, which can be data integrity, code integrity, data-flow integrity or control-flow integrity [199]. For example, data-execution prevention [192] is an example of protecting data integrity. In this framework, IDST maintains data-flow integrity by pointer rewriting and instrumented bounds checking.

Improving code coverage is an important problem in software testing. Two main approaches have emerged from this research, namely automated test-case generation and control-flow modification. The goal of the former approach is to generate test cases to maximise the code covered. Examples of this approach include random testing or fuzzing [200, 201], symbolic execution, e.g. [202], and concolic execution [203, 194, 177]. On the other hand, control-flow modification takes a different approach by forcibly altering execution flow at one or more conditional branches, for example to skip calculation of checksums which are expensive to solve symbolically [195, 204]. Control-flow modification so far has only been applied in a limited number of contexts. This chapter proposes probabilistic branch negation as a technique for aiding type recovery and decompilation in general. The average type distance was optimal when $P_b = 0.2$, implying that the code coverage is maximised under this condition. Further work is needed to understand this phenomenon better.

7

Rendezvous: a prototype search engine

The token-based approach, described in Chapter 4, was implemented as a search engine, currently hosted at `http://www.rendezvousalpha.com`. This chapter discusses its architecture, results, storage requirements and performance.

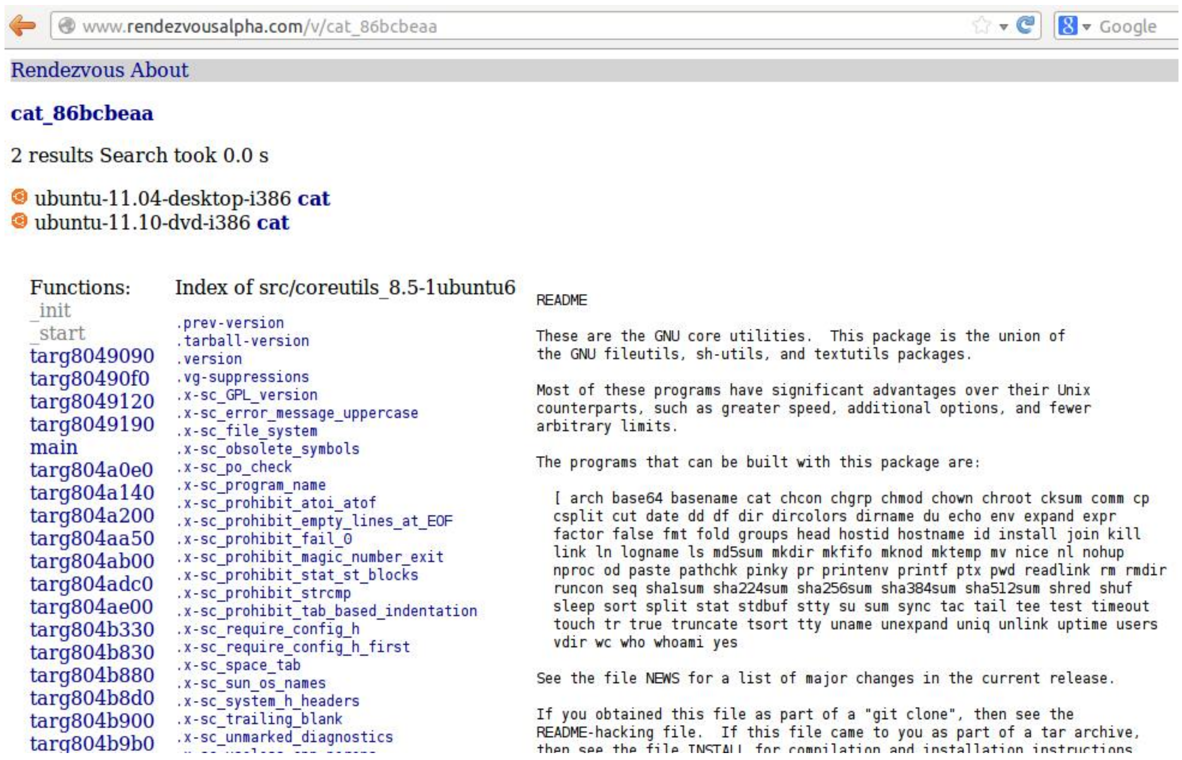
7.1 System architecture

The front-end was implemented in 1,655 lines of GO; the back-end comprised a disassembler, which relied on the Dyninst binary instrumentation tool and the text search engine CLucene. As a proof-of-concept prototype, code indexing in Rendezvous was focussed on 32-bit x86 executables in the Ubuntu 11.04 base distribution using instruction mnemonic 2- and 4-grams.

The matching process in Rendezvous was performed at the binary level, i.e. the binaries in the corpus were matched against the binary under consideration. The mapping of the search results back to the source was done using the `-aux-info <output>` option in `gcc`, which provided the locations of all function declarations in the source code. However, `-aux-info <output>` overwrites the output file with every call to `gcc`. Since the output file for logging the auxiliary information is the same for all source files, that information is lost with every new file compiled. To overcome this issue, `gcc` was modified so that the call to `fopen` for the `-aux-info` option was append (a) rather than write (w).

7.2 Results and performance

Two levels of search were implemented: queries could be made at either the executable level or the function level. Executable-level queries are automatically made when the binary is uploaded to the Rendezvous server. The results page for executable-level searches displays the ranked results list in the top pane, with the bottom pane showing the list of functions on the left, the list of files in the source package, if any, in the centre and the contents of the `README` file, if any, on the right (Figure 7.1). This example shows two results returned, namely `cat` from the 11.04 and 11.10 Ubuntu distributions.



www.rendezvousalpha.com/v/cat_86bcbeaa

Rendezvous About

cat_86bcbeaa

2 results Search took 0.0 s

- ubuntu-11.04-desktop-i386 **cat**
- ubuntu-11.10-dvd-i386 **cat**

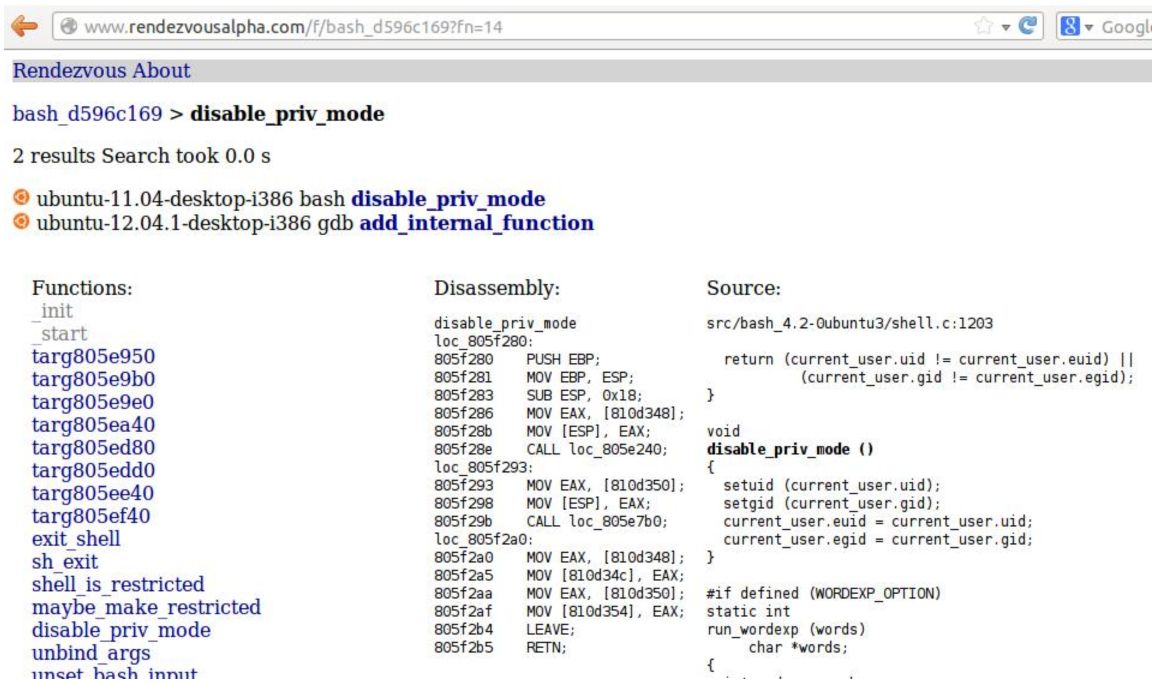
Functions:
_init
_start
targ8049090
targ80490f0
targ8049120
targ8049190
main
targ804a0e0
targ804a140
targ804a200
targ804aa50
targ804ab00
targ804adc0
targ804ae00
targ804b330
targ804b830
targ804b880
targ804b8d0
targ804b900
targ804b9b0

Index of src/coreutils_8.5-1ubuntu6

```
.prev-version
.tarball-version
.version
.vg-suppressions
.x-sc_GPL_version
.x-sc_error_message_uppercase
.x-sc_file_system
.x-sc_obsolete_symbols
.x-sc_po_check
.x-sc_program_name
.x-sc_prohibit_atoi_atof
.x-sc_prohibit_empty_lines_at_EOF
.x-sc_prohibit_fail_0
.x-sc_prohibit_magic_number_exit
.x-sc_prohibit_stat_st_blocks
.x-sc_prohibit_strcmp
.x-sc_prohibit_tab_based_indentation
.x-sc_require_config_h
.x-sc_require_config_h_first
.x-sc_space_tab
.x-sc_sun_os_names
.x-sc_system_h_headers
.x-sc_trailing_blank
.x-sc_unmarked_diagnostics
```

```
README

These are the GNU core utilities.  This package is the union of
the GNU fileutils, sh-utils, and textutils packages.

Most of these programs have significant advantages over their Unix
counterparts, such as greater speed, additional options, and fewer
arbitrary limits.

The programs that can be built with this package are:

  [ arch base64 basename cat chcon chgrp chmod chown chroot cksum comm cp
  csplit cut date dd df dir dircolors dirname du echo env expand expr
  factor false fmt fold groups head hostid hostname id install join kill
  link ln logname ls md5sum mkdir mkfifo mknod mktemp mv nice nl nohup
  nproc od paste pathchk pinky pr printenv printf ptx pwd readlink rm rmdir
  runcon seq sha1sum sha224sum sha256sum sha384sum sha512sum shred shuf
  sleep sort split stat stdbuf stty su sum sync tac tail tee test timeout
  touch tr true truncate tsort tty uname unexpand uniq unlink uptime users
  vdir wc who whoami yes

See the file NEWS for a list of major changes in the current release.

If you obtained this file as part of a "git clone", then see the
README-hacking file.  If this file came to you as part of a tar archive,
then see the file INSTALL for compilation and installation instructions
```

Figure 7.1: The results page for an executable-level query for `cat`.



www.rendezvousalpha.com/f/bash_d596c169?fn=14

Rendezvous About

bash_d596c169 > **disable_priv_mode**

2 results Search took 0.0 s

- ubuntu-11.04-desktop-i386 bash **disable_priv_mode**
- ubuntu-12.04.1-desktop-i386 gdb **add_internal_function**

Functions:
_init
_start
targ805e950
targ805e9b0
targ805e9e0
targ805ea40
targ805ed80
targ805edd0
targ805ee40
targ805ef40
exit_shell
sh_exit
shell_is_restricted
maybe_make_restricted
disable_priv_mode
unbind_args
unset_bash_input

Disassembly:

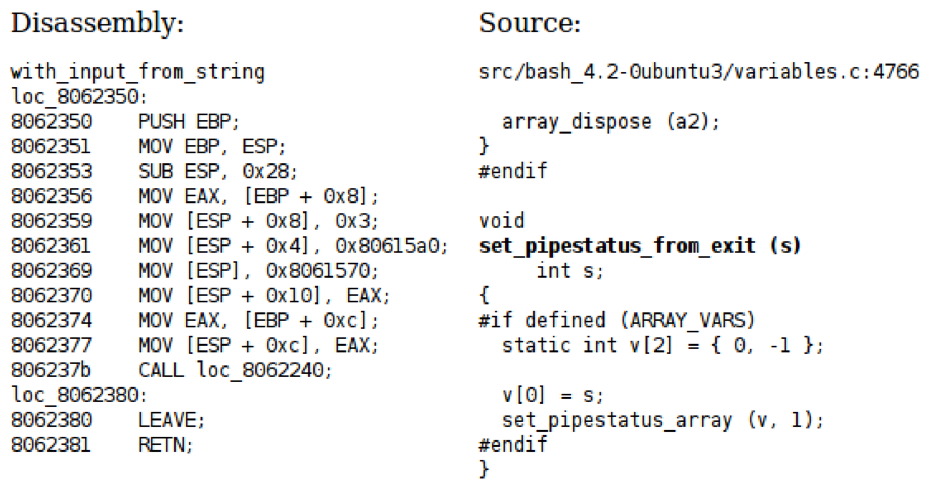
```
disable_priv_mode
loc_805f280:
805f280     PUSH EBP;
805f281     MOV EBP, ESP;
805f283     SUB ESP, 0x18;
805f286     MOV EAX, [810d348];
805f28b     MOV [ESP], EAX;
805f28e     CALL loc_805e240;
loc_805f293:
805f293     MOV EAX, [810d350];
805f298     MOV [ESP], EAX;
805f29b     CALL loc_805e7b0;
loc_805f2a0:
805f2a0     MOV EAX, [810d348];
805f2a5     MOV [810d34c], EAX;
805f2aa     MOV EAX, [810d350];
805f2af     MOV [810d354], EAX;
805f2b4     LEAVE;
805f2b5     RETN;
```

Source:

```
src/bash_4.2-0ubuntu3/shell.c:1203

  return (current_user.uid != current_user.euid) ||
          (current_user.gid != current_user.egid);
}

void
disable_priv_mode ()
{
  setuid (current_user.uid);
  setgid (current_user.gid);
  current_user.euid = current_user.uid;
  current_user.egid = current_user.gid;
}

#if defined (WORDEXP_OPTION)
static int
run_wordexp (words)
     char *words;
{
```

Figure 7.2: The results page for a function-level query for `disable_priv_mode`.

Function-level queries are embedded as links in the function list in the bottom left pane of both types of results page. The results page for function-level searches similarly displays the ranked results in the top pane and the function list in the bottom left. The centre pane displays the disassembly of the function under consideration and the right pane the source code of the chosen search result, if any (Figure 7.2). The search result for the



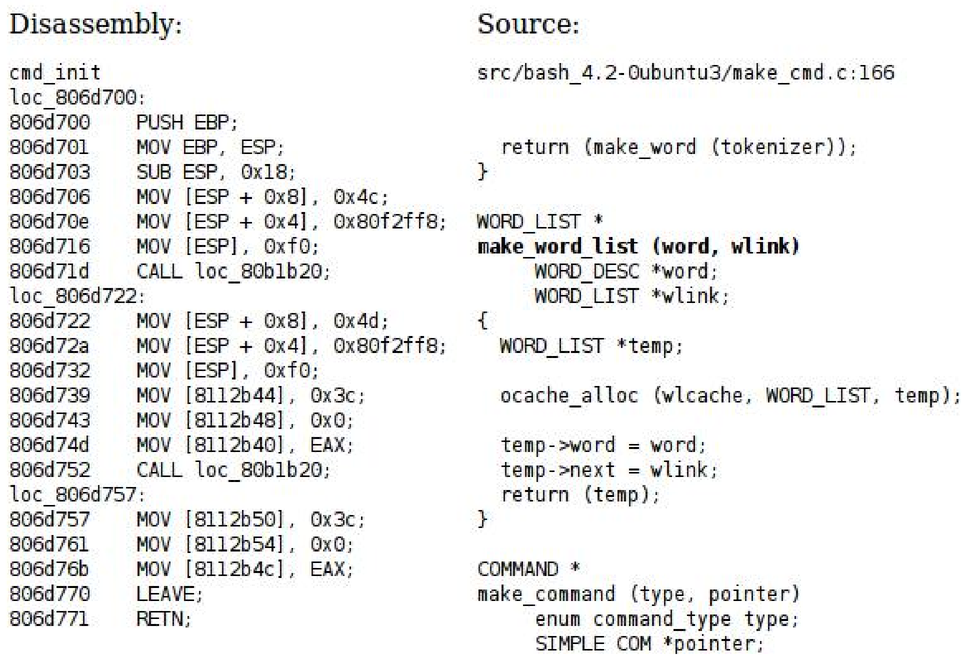
Disassembly:

```
with_input_from_string
loc_8062350:
8062350     PUSH EBP;
8062351     MOV EBP, ESP;
8062353     SUB ESP, 0x28;
8062356     MOV EAX, [EBP + 0x8];
8062359     MOV [ESP + 0x8], 0x3;
8062361     MOV [ESP + 0x4], 0x80615a0;
8062369     MOV [ESP], 0x8061570;
8062370     MOV [ESP + 0x10], EAX;
8062374     MOV EAX, [EBP + 0xc];
8062377     MOV [ESP + 0xc], EAX;
806237b     CALL loc_8062240;
loc_8062380:
8062380     LEAVE;
8062381     RETN;
```

Source:

```
src/bash_4.2-0ubuntu3/variables.c:4766

  array_dispose (a2);
}
#endif

void
set_pipestatus_from_exit (s)
     int s;
{
#if defined (ARRAY_VARS)
  static int v[2] = { 0, -1 };

  v[0] = s;
  set_pipestatus_array (v, 1);
#endif
}
```

Figure 7.3: Function under consideration, `with_input_from_string` (left), with the closest match `set_pipestatus_from_exit` (right).



Disassembly:

```
cmd_init
loc_806d700:
806d700     PUSH EBP;
806d701     MOV EBP, ESP;
806d703     SUB ESP, 0x18;
806d706     MOV [ESP + 0x8], 0x4c;
806d70e     MOV [ESP + 0x4], 0x80f2ff8;
806d716     MOV [ESP], 0xf0;
806d71d     CALL loc_80b1b20;
loc_806d722:
806d722     MOV [ESP + 0x8], 0x4d;
806d72a     MOV [ESP + 0x4], 0x80f2ff8;
806d732     MOV [ESP], 0xf0;
806d739     MOV [8112b44], 0x3c;
806d743     MOV [8112b48], 0x0;
806d74d     MOV [8112b40], EAX;
806d752     CALL loc_80b1b20;
loc_806d757:
806d757     MOV [8112b50], 0x3c;
806d761     MOV [8112b54], 0x0;
806d76b     MOV [8112b4c], EAX;
806d770     LEAVE;
806d771     RETN;
```

Source:

```
src/bash_4.2-0ubuntu3/make_cmd.c:166


  return (make_word (tokenizer));
}

WORD_LIST *
make_word_list (word, wlink)
     WORD_DESC *word;
     WORD_LIST *wlink;
{
  WORD_LIST *temp;

  ocache_alloc (wlcache, WORD_LIST, temp);

  temp->word = word;
  temp->next = wlink;
  return (temp);
}

COMMAND *
make_command (type, pointer)
     enum command_type type;
     SIMPLE_COM *pointer;
```

Figure 7.4: Function under consideration, `cmd_init` (left), with the closest match `make_word_list` (right).

`disable_priv_mode` function lists the correct match as the first entry.

Based on a random sample of thirty queries, the precision and recall rates were 40.7% and 88.9% respectively, giving an F_2-measure of 71.9%. This is in the same ballpark as the results predicted by Table 4.5. Exact matches, or instances where the correct match was the only one returned, occurred 63.0% of the time.

Occasionally where inexact matches were returned by Rendezvous, these functions were still useful from an analyst's perspective since they bore some resemblance to the query function. For example, the closest solution for `with_input_from_string` was `set_pipestatus_from_exit` which, on manual inspection, was not an exact match (Figure 7.3).

	Average (s)	Worst (s)
Disassembly	0.773	0.827
CLucene query	0.148	1.317
Total (results uncached)	2.30	9.88
Total (results cached)	0.381	0.964

Table 7.1: Time taken for the back-end disassembly and query phases, and for uncached and cached timings including the front-end.

Nevertheless, the higher-order behaviour was similar: the initialisation of a structure which is passed as a parameter to a callee function.

Another example of an incorrect match was the query for `cmd_init` which returned `make_word_list` as the closest matched function (Figure 7.4). Here too, the higher-order behaviour was similar: a function call followed by writing to a data structure and returning, although there was disagreement as to whether one or two calls were made. This could in theory be resolved by including 5-grams, since, assuming the mnemonic sequence for `make_word_list` was `PUSH, MOV, SUB, MOV, MOV, MOV, CALL, MOV, MOV, MOV, LEAVE, RETN`, the presence of the 5-gram `CALL, MOV, MOV, MOV, MOV` in `cmd_init` would eliminate `make_word_list` as a candidate.

Although an exact match was not always obtained, a partial match provides the basis for further refinement, and is a topic for further research (see Future work).

7.2.1 Storage requirements

A total of 40,929 executables from five distributions—Ubuntu 11.04, Ubuntu 11.10, FreeBSD 9.0, Debian 5.0.5 and Fedora 18—and 192,239 functions from executables in the Ubuntu 11.04 base distribution were indexed by Rendezvous. The "executables" indices required 5.6 GB and the "functions" index 126 MB. In all, the size of the server was a manageable 11 GB, with the CLucene indices taking up half of the disk space and the source code of Ubuntu 11.04 taking up the other half.

CLucene initially generated an `EFBIG` (File too large) error when trying to generate an index that was larger than 2 GB, due to the 2 GB file size limit imposed in 32-bit Linux. This was overcome by generating multiple CLucene indices and having the front-end combine the results of multiple queries. Porting the server to a 64-bit operating system should solve this issue.

7.2.2 Performance

The time taken to generate the six CLucene indices on an Intel Duo Core 2 machine with 1 GB of RAM was about two hours. The current version of Rendezvous takes between 1–10 seconds to disassemble, query and return if the results are uncached. Otherwise, requests are handled within 0.3–0.9 seconds. The disassembly phase is the main bottleneck at the moment, consistently taking between 0.7–0.9 seconds (Table 7.1) and improving the disassembler is a topic for future work.

8

Conclusions and future work

Decompilation has been viewed as trying to reconstruct "pigs from sausages" [51]. On the other hand, the need for reverse engineering tools has increased and the research community has responded with better tools and techniques. The goals of decompilation can be summarised as producing compilable, equivalent and readable code. Current tools focus on one or two of these goals at a time.

This thesis has argued for search-based decompilation (SBD) based on empirical evidence that code reuse is prevalent. With SBD, the goals of obtaining compilable, equivalent and readable code are simultaneously met when a suitable match is found. Two techniques were proposed. The first, token-based code indexing, was shown to scale to at least a real-world corpus consisting of 40,929 executables and 192,239 functions, having an F_2-measure of 83.0–86.7% for programs compiled with two different configurations.

The second, test- and perturbation-based indexing, was shown to have high precision and recall (88.4% F_2-measure) for a smaller data set of internal functions compiled with five different configurations. The effectiveness of perturbation analysis was demonstrated by its ability to distinguish between eleven variants of five sorting algorithms. The main task ahead for TBI/PBI is in dealing with API and system calls.

Prototype recovery is a prerequisite for perturbation analysis, and the proposed approach, inlined data source tracking with probabilistic branch negation, was demonstrated to have a 62–84% success rate on a dataset of 443 leaf functions. Its accuracy (0.175 false positive rate) was higher than the current state-of-the-art prototype-recovery algorithm in terms of inferring the correct number of parameters.

8.1 Future work

Work on the SBD approach is certainly by no means complete, and SBD can benefit from further research in the following areas.

Development of decompilation evaluation metrics Metrics are an important aspect of SBD, and indeed for decompilation research on the whole. At the moment, evaluation centres around comparisons with the output of existing decompilers, some of which are proprietary or closed-source. Having a corpus of source and binary programs is helpful, but an evaluation metric that can add finer-grained details, such as source-binary variable alignment, will be more beneficial. Perturbation analysis (Chapter 5) is a possible candidate, but more work needs to be done to address the handling of system and API calls.

Sparse/Specialised disassembly As noted in Chapter 4 and in the Rendezvous prototype, disassembly is currently a performance bottle-neck, especially for large binaries. Possible research questions to ask here are: what is the trade-off like in terms of performing complete disassembly versus a "sparse" disassembly? What specialised disassembly methods can/should be adopted for lightly obfuscated binaries, e.g. XOR-encrypted or UPX-packed[1] binaries?

Partial matching Thus far the primary consideration has been function-level matching—either there is a match for the whole function or not at all. What is to be done in cases where a function is partially modified from the original? One possible approach is to first find the closest match, then patch the source code until it is an exact match.

This thesis has demonstrated beyond a reasonable doubt that search can be an enormous enabler in decompiling programs that are getting larger, more complex and reusing more software. The challenge now is to transition from a prototype to a production system, and scaling up to index and match more software. That is a matter of engineering; the feasibility of search-based decompilation is no longer in question.

[1]`http://upx.sourceforge.net`

Bibliography

[1] K. Thompson, “Reflections on trusting trust,” *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984.

[2] D. A. Wheeler, “Fully countering Trusting Trust through diverse double-compiling,” Ph.D. dissertation, George Mason University, 2009.

[3] K. Zetter, “How digital detectives deciphered Stuxnet,” http://www.wired.com/threatlevel/2011/07/how-digital-detectives-deciphered-stuxnet/, 2011.

[4] F. Lindner, “Targeted industrial control system attacks,” in *Hashdays security conference*, 2011.

[5] G. Balakrishnan and T. Reps, “WYSINWYX: What you see is not what you execute,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010.

[6] A. Cox and T. Smedley, “Reverse engineering in support of litigation: Experiences in an adversarial environment,” in *Proceedings of the 13th Working Conference on Reverse Engineering*, ser. WCRE ’06. IEEE Computer Society, 2006, pp. 288–292.

[7] Hex-Rays SA, “The Hex-Rays decompiler,” http://www.hex-rays.com/products/decompiler/index.shtml.

[8] “Boomerang decompiler,” http://boomerang.sourceforge.net.

[9] B. Schwarz, S. K. Debray, and G. R. Andrews, “Disassembly of executable code revisited,” in *Proceedings of the 2002 9th Working Conference on Reverse Engineering*, ser. WCRE ’02, A. van Deursen and E. Burd, Eds. IEEE Computer Society, 2002, pp. 45–54.

[10] J. Kinder, F. Zuleger, and H. Veith, “An abstract interpretation-based framework for control flow reconstruction from binaries,” in *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 214–228.

[11] C. Cifuentes, “Reverse compilation techniques,” Ph.D. dissertation, University of Queensland, 1994.

[12] A. Mycroft, “Type-based decompilation (or program reconstruction via type reconstruction),” in *Proceedings of the 8th European Symposium on Programming*, ser. ESOP ’99, S. D. Swierstra, Ed. Springer, 1999, pp. 208–223.

[13] M. J. Van Emmerik, “Static single assignment for decompilation,” Ph.D. dissertation, University of Queensland, 2007.

[14] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, T. Zimmermann, M. D. Penta, and S. Kim, Eds. IEEE / ACM, May 2013, pp. 329–338.

[15] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "SmartDec: Approaching C++ decompilation," in *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, ser. WCRE '11, M. Pinzger, D. Poshyvanyk, and J. Buckley, Eds. Washington, DC, USA: IEEE Computer Society, 2011, pp. 347–356.

[16] M. H. Halstead, "Machine-independent computer programming," *Chapter 11*, pp. 143–150, 1962.

[17] M. H. Halstead, "Machine-independence and third-generation computers," in *AFIPS Fall Joint Computing Conference*, 1967, pp. 587–592.

[18] W. A. Sassaman, "A computer program to translate machine language into FORTRAN," in *American Federation of Information Processing Societies: Proceedings of the Joint Computer Conference*, ser. AFIPS '66 (Spring). New York, NY, USA: ACM, April 1966, pp. 235–239.

[19] C. R. Hollander, "Decompilation of object programs," Ph.D. dissertation, Stanford University, 1973.

[20] V. Schneider and G. Winiger, "Translation grammars for compilation and decompilation," *BIT Numerical Mathematics*, vol. 14, no. 1, pp. 78–86, 1974.

[21] S. T. Hood, "Decompiling with definite clause grammars," Defence Science and Technology Organisation, Electronics Research Laboratory, Salisbury, South Australia, Tech. Rep. ERL-0571-RR, September 1991.

[22] J. J. O'Gorman, "Systematic decompilation," Ph.D. dissertation, University of Limerick, 1991.

[23] F. Chen and Z. Liu, "C function recognition technique and its implementation in 8086 C decompiling system," *Mini-Micro Systems*, vol. 12, no. 11, pp. 33–40,47, 1991.

[24] F. Chen, Z. Liu, and L. Li, "Design and implementation techniques of the 8086 C decompiling system," *Mini-Micro Systems*, vol. 14, no. 4, pp. 10–18,31, 1993.

[25] S. Kumar, "DisC - decompiler for TurboC," http://www.debugmode.com/dcompile/disc.htm, 2001.

[26] I. Guilfanov, "Simple type system for program reengineering," in *Proceedings of the 8th Working Conference on Reverse Engineering*, ser. WCRE '01. Los Alamitos, CA, USA: IEEE Computer Society, 2001, pp. 357–361.

[27] I. Guilfanov, "Decompilation gets real," http://www.hexblog.com/?p=56.

[28] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of C++ programs," in *CSMR*, 2010, pp. 240–243.

[29] E. N. Dolgova and A. V. Chernov, "Automatic reconstruction of data types in the decompilation problem," *Programming and Computer Software*, vol. 35, no. 2, pp. 105–119, 2009.

[30] B. C. Housel, "A study of decompiling machine languages into high-level machine independent languages," Ph.D. dissertation, Purdue University, 1973.

[31] B. C. Housel and M. H. Halstead, "A methodology for machine language decompilation," in *Proceedings of the 1974 annual conference - Volume 1*, ser. ACM '74. New York, NY, USA: ACM, 1974, pp. 254–260.

[32] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Softw. Pract. Exper.*, vol. 25, no. 7, pp. 811–829, Jul. 1995.

[33] C. Cifuentes, "Interprocedural data flow decompilation," *J. Prog. Lang.*, vol. 4, no. 2, pp. 77–99, 1996.

[34] A. Johnstone, E. Scott, and T. Womack, "Reverse compilation for digital signal processors: A working example," in *Proceedings of the 33rd Hawaii International Conference on System Sciences - Volume 8*, ser. HICSS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 8003–.

[35] C. Cifuentes and M. V. Emmerik, "Recovery of jump table case statements from binary code," in *7th International Workshop on Program Comprehension*, ser. IWPC '99. IEEE Computer Society, 1999, pp. 192–199.

[36] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, ser. NDSS '11. The Internet Society, 2011.

[37] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '77, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. New York, NY, USA: ACM, 1977, pp. 238–252.

[38] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proceedings of the 13th International Conference on Compiler Construction*, 2004.

[39] G. Balakrishnan and T. W. Reps, "DIVINE: Discovering variables in executables," in *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI '07, B. Cook and A. Podelski, Eds., vol. 4349. Springer, 2007, pp. 1–28.

[40] T. W. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *Proceedings of the 17th International Conference on Compiler Construction*, ser. CC '08, L. J. Hendren, Ed., vol. 4959. Springer, 2008, pp. 16–35.

[41] T. W. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum, "A next-generation platform for analyzing executables," in *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, ser. APLAS '05, K. Yi, Ed., vol. 3780. Springer, 2005, pp. 212–229.

[42] B.-Y. E. Chang, M. Harren, and G. C. Necula, “Analysis of low-level code using cooperating decompilers,” in *Proceedings of the 13th international conference on Static Analysis*, ser. SAS’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 318–335.

[43] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong progrm analysis & transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, 2004.

[44] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, and R. Barua, “Decompilation to high IR in a binary rewriter,” University of Maryland, Tech. Rep., 2010.

[45] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, “A compiler-level intermediate representation based binary analysis and rewriting system,” in *EuroSys*, 2013, pp. 295–308.

[46] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, “Scalable variable and data type detection in a binary rewriter,” in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’13, H.-J. Boehm and C. Flanagan, Eds. New York, NY, USA: ACM, 2013, pp. 51–60.

[47] J. P. Bowen, “From programs to object code using logic and logic programming,” in *Code Generation – Concepts, Tools, Techniques, Proc. International Workshop on Code Generation, Dagstuhl.* Springer-Verlag, 1992, pp. 173–192.

[48] J. Bowen, “From programs to object code and back again using logic programming: Compilation and decompilation,” *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 205–234, 1993.

[49] P. T. Breuer and J. P. Bowen, “Decompilation: The enumeration of types and grammars,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1613–1647, Sep. 1994.

[50] M. P. Ward, “Reverse engineering from assembler to formal specifications via program transformations,” in *Proceedings of the 7th Working Conference on Reverse Engineering*, ser. WCRE ’00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 11–.

[51] M. P. Ward, “Pigs from sausages? Reengineering from assembler to C via FermaT transformations,” *Science of Computer Programming Special Issue: Transformations Everywhere*, vol. 52, no. 1–3, pp. 213–255, 2004.

[52] S. Katsumata and A. Ohori, “Proof-directed de-compilation of low-level code,” in *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ser. ESOP ’01, D. Sands, Ed. London, UK: Springer-Verlag, 2001, pp. 352–366.

[53] A. Mycroft, A. Ohori, and S. Katsumata, “Comparing type-based and proof-directed decompilation,” in *Proceedings of the 2001 8th Working Conference on Reverse Engineering*, ser. WCRE ’01. IEEE Computer Society, 2001, pp. 362–367.

[54] M. Myreen, M. Gordon, and K. Slind, "Machine-code verification for multiple architectures - an application of decompilation into logic," in *Formal Methods in Computer-Aided Design*, ser. FMCAD '08, 2008, pp. 1–8.

[55] M. Myreen, M. Gordon, and K. Slind, "Decompilation into logic – improved," in *Formal Methods in Computer-Aided Design*, ser. FMCAD '12, 2012, pp. 78–81.

[56] N. Ramsey and M. F. Fernandez, "The New Jersey machine-code toolkit," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON '95. Berkeley, CA, USA: USENIX Association, 1995, pp. 24–24.

[57] C. Cifuentes, M. V. Emmerik, and N. Ramsey, "The design of a resourceable and retargetable binary translator," in *Proceedings of the 1999 6th Working Conference on Reverse Engineering*, ser. WCRE '99. IEEE Computer Society, 1999, pp. 280–291.

[58] C. Cifuentes and M. V. Emmerik, "UQBT: Adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, pp. 60–66, Mar. 2000.

[59] J. Tröger and C. Cifuentes, "Analysis of virtual method invocation for binary translation," in *Proceedings of the 2002 9th Working Conference on Reverse Engineering*, ser. WCRE '02, A. van Deursen and E. Burd, Eds. IEEE Computer Society, 2002, pp. 65–.

[60] N. Ramsey and C. Cifuentes, "A transformational approach to binary translation of delayed branches," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 2, pp. 210–224, 2003.

[61] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with RevNIC," in *EuroSys*, 2010, pp. 167–180.

[62] V. Chipounov and G. Candea, "Enabling sophisticated analyses of x86 binaries with RevGen," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, 2011, pp. 211–216.

[63] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, ser. NDSS '10. The Internet Society, 2010.

[64] B. Feigin and A. Mycroft, "Decompilation is an information-flow problem," in *Proceedings of the 4th International Workshop on Programming Language Interference and Dependence (PLID 2008)*, Valencia, Spain, July 2008.

[65] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, ser. POPL '09, Z. Shao and B. C. Pierce, Eds. New York, NY, USA: ACM, 2009, pp. 264–276.

[66] B. Feigin, "Interpretational overhead in system software," Ph.D. dissertation, University of Cambridge, 2011.

[67] D. Vermoen, M. Witteman, and G. N. Gaydadjiev, "Reverse engineering Java card applets using power analysis," in *Proceedings of the 1st IFIP TC6 /WG8.8 /WG11.2 international conference on Information security theory and practices: smart cards, mobile and ubiquitous computing systems*, ser. WISTP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 138–149.

[68] T. Eisenbarth, C. Paar, and B. Weghenkel, "Building a side channel based disassembler," *Transactions on Computational Science*, vol. 10, pp. 78–99, 2010.

[69] M. D. Preda and R. Giacobazzi, "Semantic-based code obfuscation by abstract interpretation," in *ICALP*, 2005, pp. 1325–1336.

[70] M. V. Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *Proceedings of the 11th Working Conference on Reverse Engineering*, ser. WCRE '04. IEEE Computer Society, November 2004, pp. 27–36.

[71] R. P. Díaz, "Status report: Software reusability," *IEEE Software*, vol. 10, no. 3, pp. 61–66, 1993.

[72] J. C. Knight and M. F. Dunn, "Software quality through domain-driven certification," *Ann. Softw. Eng.*, vol. 5, pp. 293–315, Jan. 1998.

[73] T. Ravichandran and M. A. Rothenberger, "Software reuse strategies and component markets," *Commun. ACM*, vol. 46, no. 8, pp. 109–114, Aug. 2003.

[74] B. Meyer, *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.

[75] B. Jalender, A. Govardhan, and P. Premchand, "A pragmatic approach to software reuse," *J. Theoretical and Applied Information Technology*, vol. 14, no. 2, pp. 87–96, 2005.

[76] M. E. Fayad and D. C. Schmidt, "Lessons learned building reusable OO frameworks for distributed software," *Commun. ACM*, vol. 40, no. 10, pp. 85–87, Oct. 1997.

[77] D. C. Schmidt, "Why software reuse had failed and how to make it work for you," 2006. [Online]. Available: http://www.cs.westl.edu/~schmidt/reuse-lessons.html [Accessed: 4/3/2013]

[78] S. Haefliger, G. von Krogh, and S. Spaeth, "Code reuse in open source software," *Management Science*, vol. 54, no. 1, pp. 180–193, January 2008.

[79] J. S. Poulin, "Populating software repositories: Incentives and domain-specific software," *J. Syst. Softw.*, vol. 30, no. 3, pp. 187–199, Sep. 1995.

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

[81] M. A. Cusumano, *Japan's software factories: A challenge to U.S. management.* New York, NY, USA: Oxford University Press, Inc., 1991.

[82] GNU Foundation, "GNU Public License version 2," http://www.gnu.org/licenses/gpl-2.0.html, June 1991.

[83] Software Freedom Law Center, "Software Freedom Law Center," http://www.softwarefreedom.org.

[84] Free Software Foundation, Inc., "Free software foundation," http://www.fsf.org.

[85] H. Welte, "Current Developments in GPL Compliance," http:taipei.freedomhec.org/dlfile/gpl_compliance.pdf, June 2012, freedomHEC Taipei 2012 conference.

[86] B. Kuhn, "GPL enforcement: Don't jump to conclusions, but do report violations," http://ebb.org/bkuhn/blog/2009/11/08/gpl-enforcement.html, 11 2009.

[87] "Progress Software Corporation v MySQL AB," February 2002, 195 F. Supp. 2d 238, D. Mass. [Online]. Available: http://www.leagle.com/xmlResult.aspx?xmldoc=2002523195FSupp2d238_1505.xml [Accessed: 27/02/2013]

[88] H. Welte, "Allnet GmbH resolves iptables GPL violation," February 2004. [Online]. Available: http://lwn.net/Articles/71418 [Accessed: 28/2/2013]

[89] H. Welte, "Netfilter/iptables project concludes agreement on GPL licensing with Gigabyte," October 2004. [Online]. Available: http://www.gpl-violations.org/news/20041022-iptables-gigabyte.html [Accessed: 27/2/2013]

[90] "Welte v. Sitecom Deutschland GmbH," March 2004, District Court of Munich, case 21 O 6123/04. [Online]. Available: http://www.jbb.de/fileadmin/download/urteil_lg_muenchen_gpl.pdf [Accessed: 28/2/2013]

[91] "Welte v. Sitecom Deutschland GmbH," March 2004, District Court of Munich, case 21 O 6123/04 (English translation). [Online]. Available: http://www.jbb.de/fileadmin/download/judgement_dc_munich_gpl.pdf [Accessed: 28/2/2013]

[92] H. Welte, "Gpl-violations.org project was granted a preliminary injunction against Fortinet UK Ltd." April 2005. [Online]. Available: http://www.gpl-violations.org/news/20050414-fortinet-injunction.html [Accessed: 28/2/2013]

[93] H. Welte, "Gpl-violations.org project prevails in court case on GPL violation by D-Link," September 2006. [Online]. Available: http://www.gpl-violations.org/news/20060922-dlink-judgement_frankfurt.html [Accessed: 28/2/2013]

[94] "Welte v. D-Link," September 2006, 2-6 0 224/06, Landgericht Frankfurt AM Main. [Online]. Available: http://www.jjb.de/urteil_lg_frankfurt_gpl.pdf [Accessed: 28/2/2013]

[95] Software Freedom Law Center, Inc., "BusyBox Developers and Monsoon Multimedia agree to dismiss GPL lawsuit," October 2007. [Online]. Available: http://www.softwarefreedom.org/news/2007/oct/30/busybox-monsoon-settlement/ [Accessed: 28/2/2013]

[96] Software Freedom Law Center, Inc., "BusyBox Developers and High-Gain Antennas agree to dismiss GPL lawsuit," March 2008. [Online]. Available: http://www.softwarefreedom.org/news/2008/mar/06/busybox-hga/ [Accessed: 28/2/2013]

[97] Software Freedom Law Center, Inc., "BusyBox Developers and Xterasys Corporation agree to dismiss GPL lawsuit," December 2007. [Online]. Available: http://www.softwarefreedom.org/news/2007/dec/17/busybox-xterasys-settlement/ [Accessed: 28/2/2013]

[98] Software Freedom Law Center, Inc., "BusyBox Developers agree to end GPL lawsuit against Verizon," March 2008. [Online]. Available: http://www.softwarefreedom.org/news/2008/mar/17/busybox-verizon/ [Accessed: 28/2/2013]

[99] Software Freedom Law Center, Inc., "BusyBox developers and Supermicro agree to end GPL lawsuit," July 2008. [Online]. Available: http://www.softwarefreedom.org/news/2008/jul/23/busybox-supermicro/ [Accessed: 28/2/2013]

[100] "Andersen v. Bell Microproducts, Inc." October 2008, No. 08-CV-5270, Doc. No. 16, S.D.N.Y., notice of voluntary dismissal.

[101] "Andersen v. Bell Microproducts, Inc." June 2008, Civil Action No. 08-CV-5270. [Online]. Available: http://www.softwarefreedom.org/news/2008/jun/10/busybox/bell-complaint.pdf [Accessed: 28/2/2013]

[102] "Erik Andersen and Rob Landlet v. Super Micro Computer, Inc." June 2008, Civil Action No. 08-CV-5269. [Online]. Available: http://www.softwarefreedom.org/news/2008/jun/10/busybox/supermicro-complaint.pdf [Accessed: 28/2/2013]

[103] "Welte v. Skype Technologies S.A." July 2007, District Court of Munich, case 7 O 5245/07. [Online]. Available: http://www.ifross.de/Fremdartikel/LGMuenchenUrteil.pdf [Accessed: 28/2/2013]

[104] "Free Software Foundation, Inc. v. Cisco Systems, Inc." December 2008, Civil Action No. 08-CV-10764. [Online]. Available: http://www.fsf.org/licensing/complaint-2008-12-11.pdf [Accessed: 28/2/2013]

[105] B. Smith, "FSF settles suit against Cisco," May 2009. [Online]. Available: http://www.fsf.org/news/2009-05-cisco-settlement [Accessed: 28/2/2013]

[106] J. Klinger, "Jin vs. IChessU: The copyright infringement case (settled)," October 2008. [Online]. Available: http://www.jinchess.com/ichessu/ [Accessed: 28/2/2013]

[107] "Software Freedom Conservancy, Inc. and Erik Andersen v. Best Buy Co., Inc. et al." December 2009, Civil Action No. 09-CV-10155. [Online]. Available: http://www.softwarefreedom.org/resources/2009/busybox-compliant-2009-12-14.pdf [Accessed: 27/2/2013]

[108] "Software Freedom Conservancy, Inc. and Erik Andersen v. Best Buy Co., Inc. et al." http://www.softwarefreedom.org/resources/2010/2010-06-03-Motion_against_Westinghouse.pdf, June 2010, No. 09-CV-10155, Motion for default judgment against Westinghouse Digital Electronics LLC. and associated documents.

[109] M. von Willebrand, "A look at EDU 4 v. AFPA, also known as the "Paris GPL case"," *IFOSS L. Rev.*, vol. 1, no. 2, pp. 123–126, 2009.

[110] "AVM Computersysteme Vertriebs GmbH v. Bybits AG," November 2011, 16 O 255/10, Landgericht Berlin.

[111] P. Galli, “Free Software Foundation targets RTLinux for GPL violations,” September 2011. [Online]. Available: http://www.eweek.com/c/a/Application-Development/Free-Software-Foundation-Targets-RTLinux-for-GPL-violations/ [Accessed: 28/2/2013]

[112] T. Gasperson, “FSF: Lindows OS moving toward GPL compliance,” June 2022. [Online]. Available: http://archive09.linux.com/articles/23277 [Accessed: 28/2/2013]

[113] S. Shankland, “GPL gains clout in German legal case,” April 2044. [Online]. Available: http://news.cnet.com/2100-7344-5198117.html [Accessed: 1/3/2013]

[114] H. Welte, “13 Companies at CeBIT receive warning letter regarding their alleged GPL incompliance,” March 2005. [Online]. Available: http://gol-violations.org/news/20050314-cebit-letter-action.html [Accessed: 1/3/2013]

[115] R. Rohde, “Rohde v. Viasat on GPL/LGPL,” 2009. [Online]. Available: http://duff.dk/viasat [Accessed: 28/2/2013]

[116] E. Sandulenko, “GPL, ScummVM and violations,” June 2009. [Online]. Available: http://sev-notes.blogspot.co.uk/2009/06/gpl-scummvm-and-violations.html [Accessed: 28/2/2013]

[117] S. Olsen, “Cadence, Avast settle trade-secret suit,” November 2002. [Online]. Available: http://news.cnet.com/Cadence,-Avast-settle-trade-secret-suit/2100-1025_3-965890.html [Accessed: 1/3/2013]

[118] W. Curtis, “Green Hills Software collects $4.25 million from Microtec Research to settle trade secret lawsuit,” July 1993. [Online]. Available: http://www.ghs.com/news/archive/01jul93.html [Accessed: 1/3/2013]

[119] “Cisco Systems v. Huawei Technologies,” January 2003, Civil Action No. 2-03C-027, Eastern District of Texas. [Online]. Available: http://miltest.wikispaces.com/file/view/Cisco_Huawei_Complaint.pdf [Accessed: 1/3/2013]

[120] J. Leyden, “Cisco drops Huawei lawsuit,” July 2004. [Online]. Available: http://www.theregister.co.uk/2004/07/29/cisco_huawei_case_ends [Accessed: 1/3/2013]

[121] “Compuware Corp. v. International Business Machines Corp.” December 2002, Case 2:02-CV-70906-GCS, Eastern District of Michigan. [Online]. Available: http://miltest.wikispaces.com/file/view/Compuware_IBM_Complaint.pdf [Accessed: 1/3/2013]

[122] S. Cowley and S. Larson, “IBM, Compuware reach $400M settlement,” March 2005. [Online]. Available: http://www.computerworld.com/s/article/100622/IBM_Compuware_Reach_400M_Settlement?taxonomyId=070 [Accessed: 1/3/2013]

[123] J. N. Hoover, “Symantec, Microsoft settle Windows suit,” April 2008. [Online]. Available: http://www.informationweek.co.uk/windows/operating-systems/symantec-microsoft-settle-windows-suit/207001547 [Accessed: 1/3/2013]

[124] "Veritas Operating Corp. v. Microsoft Corp." May 2006, CV06-0703, Western District of Washington. [Online]. Available: http://niltest.wikispace.com/file/view/Veritas_MSFT_Complaint.pdf [Accessed: 1/3/2013]

[125] S. Cowley, "Quest pays CA $16 million to settle lawsuit," March 2005. [Online]. Available: http://www.infoworld/com/t/busincess/quest-pays-ca-16-million-settle-lawsuit-299 [Accessed: 1/2/2013]

[126] "CA v. Rocket Software," September 2008, Case 1:07-CV-01476-ADS-MLO, Eastern District of New York. [Online]. Available: http://miltest.wikispace.com/file/view/CA_Rocket_ruling.pdf [Accessed: 1/3/2013]

[127] R. Weisman, "CA and Rocket Software reach settlement," February 2009. [Online]. Available: http://www.boston.com/business/ticker/2009/02/ca_and_rocket_s.html [Accessed: 1/3/2013]

[128] "McRoberts Software, Inc. v. Media 100, Inc." May 2003. [Online]. Available: http://miltest.wikispaces.com/file/view/McRoberts_Media100_7thCit_Op.pdf [Accessed: 1/3/2013]

[129] K. Finley, "Github has passed SourceForge and Google Code in popularity," http://readwrite.com/2011/06/02/github-has-passed-sourceforge, June 2011.

[130] J. Finkle, "Perks and paintball: Life inside a global cybercrime ring," http://www.pcpro.co.uk/features/356737/perks-and-paintball-life-inside-a-global-cybercrime-ring, 2010.

[131] J. A. Kaplan, "Software theft a problem for actual thieves," http://www.foxnews.com/tech/2010/10/01/software-theft-problem-zeus-botnet/, 2010.

[132] D. Fisher, "Zeus source code leaked," http://threatpost.com/en_us/blogs/zeus-source-code-leaked-051011, 2011.

[133] D. Goodin, "Source code leaked for pricey ZeuS crimeware kit," http://www.theregister.co.uk/2011/05/10/zeus_crimeware_kit_leaked/, 2011.

[134] S. Fewer, "Retrieving Kernel32's base address," http:/blog.harmonysecurity.com/2009_06_01_archive.html, June 2009.

[135] F. Och, "Statistical machine translation live," http://googleresearch.blogspot.co.uk/2006/04/statistical-machine-translation-live.html, 2006.

[136] W. Weaver, "Translation (1949)," in *Machine Translation of Languages.* MIT Press, 1955.

[137] P. F. Brown, J. Cocke, S. A. D. Pietra, V. J. D. Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin, "A statistical approach to machine translation," *Comput. Linguist.*, vol. 16, no. 2, pp. 79–85, Jun. 1990.

[138] A. Lopez, "Statistical machine translation," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 8:1–8:49, August 2008.

[139] M. R. Costa-Jussà, J. M. Crego, D. Vilar, J. A. R. Fonollosa, J. B. Mariño, and H. Ney, "Analysis and system combination of phrase- and n-gram-based statistical machine translation systems," in *HLT-NAACL (Short Papers)*, 2007, pp. 137–140.

[140] Y. Al-onaizan, J. Curin, M. Jahr, K. Knight, J. Lafferty, D. Melamed, F.-J. Och, D. Purdy, N. A. Smith, and D. Yarowsky, "Statistical machine translation," Final Report, JHU Summer Workshop, Tech. Rep., 1999.

[141] D. Oard and F. Och, "Rapid-response machine translation for unexpected languages," in *MT Summit IX*, 2003.

[142] P. Koehn, "Challenges in statistical machine translation," http://homepages.inf.ed.ac.uk/pkoehn/publications/challenges2004.pdf, 2004, talk given at PARC, Google, ISI, MITRE, BBN, Univ. of Montreal.

[143] A. Way and H. Hassan, "Statistical machine translation: Trend and challenges," in *Proceedings of the 2nd International Conference on Arabic Language Resources and Tools*, April 2009, keynote address.

[144] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012.

[145] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," King's College London, Department of Computer Science, London, United Kingdom, Tech. Rep. TR-09-03, 2009.

[146] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software Bertillonage: Finding the provenance of an entity," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. ACM, 2011, pp. 183–192.

[147] N. E. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? Identifying the authors of program binaries," in *Proceedings of the 16th European Symposium on Research in Computer Security*, ser. ESORICS '11, V. Atluri and C. Díaz, Eds. Springer, 2011, pp. 172–189.

[148] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ser. ISSTA '11, M. B. Dwyer and F. Tip, Eds., 2011, pp. 100–110.

[149] Z. Wang, K. Pierce, and S. McFarling, "BMAT - a binary matching tool for stale profile propagation," *J. Instruction-Level Parallelism*, vol. 2, 2000.

[150] 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.

[151] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ser. ISSTA '09, G. Rothermel and L. K. Dillon, Eds. ACM, 2009, pp. 117–128.

[152] 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*, ser. MSR ’11, A. van Deursen, T. Xie, and T. Zimmermann, Eds. IEEE, 2011, pp. 63–72.

[153] Ohloh, “The open source network,” http://www.ohloh.net/.

[154] M. Mock, “Dynamic analysis from the bottom up,” in *Proc. 1st ICSE Int. Workshop on Dynamic Analysis (WODA).* IEEE Computer Society, 2003, pp. 13–16.

[155] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Proceedings of the 21st Annual Computer Security Applications Conference*, ser. ACSAC ’07. IEEE Computer Society, Dec 2007, pp. 421–430.

[156] 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.

[157] 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.

[158] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” in *Proceedings of the 8th International Conference on Recent advances in intrusion detection*, ser. RAID’05, 2005, pp. 207–226.

[159] 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.

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

[161] 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.

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

[163] 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.

[164] 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.

[165] C. Gautier, “Software plagiarism detection with PEGs,” Master’s thesis, University of Cambridge, 2011.

[166] 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.

[167] 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.

[168] 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 USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '04. USENIX Association, 2004, pp. 20–20.

[169] 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.

[170] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE '12, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. Piscataway, NJ, USA: IEEE, 2012, pp. 364–374.

[171] 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, R. H. Zakon, Ed. New York, NY, USA: ACM, 2012, pp. 369–378.

[172] 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.

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

[174] 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*, ser. NDSS '09. The Internet Society, 2009.

[175] D. Gao, M. K. Reiter, and D. X. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *ICICS*, 2008, pp. 238–255.

[176] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ser. ISSTA '09, G. Rothermel and L. K. Dillon, Eds. ACM, 2009, pp. 81–92.

[177] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '08, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[178] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” in *ASE*, 2008, pp. 443–446.

[179] D. Kroening, E. Clarke, and K. Yorav, “Behavioral consistency of C and Verilog programs using bounded model checking,” in *Proceedings of DAC 2003*. ACM Press, 2003, pp. 368–371.

[180] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.

[181] E. Biham and A. Shamir, “Differential fault analysis: Identifying the structure of unknown ciphers sealed in tamper-proof devices,” *preprint*, 1996.

[182] R. J. Anderson and M. G. Kuhn, “Low cost attacks on tamper resistant devices,” in *Proceedings of the 5th International Workshop on Security Protocols*. London, UK, UK: Springer-Verlag, 1998, pp. 125–136.

[183] “Bochs: The open source IA-32 emulation project,” http://bochs.sourceforge.net.

[184] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: A safe dialect of C,” in *Proceedings of the General Track of the 2002 USENIX Annual Technical Conference*, ser. ATEC ’02, C. S. Ellis, Ed. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288.

[185] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “SoftBound: Highly compatible and complete spatial memory safety for C,” in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’09, M. Hind and A. Diwan, Eds. New York, NY, USA: ACM, 2009, pp. 245–258.

[186] R. Anderson, *Security Engineering, Second Edition*. Wiley, 2008.

[187] G. Myles and C. S. Collberg, “Detecting software theft via whole program path birthmarks,” in *Information Security, 7th International Conference, ISC 2004, Palo Alto, CA, USA, September 27-29, 2004, Proceedings*, ser. Lecture Notes in Computer Science, K. Zhang and Y. Zheng, Eds., vol. 3225. Springer, 2004, pp. 404–415.

[188] zynamics, “BinDiff,” http://www.zynamics.com/bindiff.html.

[189] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, ser. NDSS ’10. The Internet Society, 2010.

[190] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures,” in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, ser. NDSS ’11. The Internet Society, 2011.

[191] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller, “Detecting code reuse attacks with a model of conformant program execution (poster),” in *Proceedings of 16th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID’13, October 2013.

[192] A. van de Ven, "Limiting buffer overflows with ExecShield," http://www.redhat.com/magazine/009jul05/features/execshield/, 2005.

[193] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.

[194] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05, V. Sarkar and M. W. Hall, Eds. New York, NY, USA: ACM, 2005, pp. 213–223.

[195] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *Proceedings of the first USENIX workshop on Offensive Technologies*, ser. WOOT '07. Berkeley, CA, USA: USENIX Association, 2007, pp. 1:1–1:9.

[196] ISO, *ISO/IEC 9899-1999: Programming Languages — C.* International Organization for Standardization, Dec 1999.

[197] D. Qi, W. N. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury, "Modeling software execution environment," in *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, ser. WCRE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 415–424.

[198] J. Zhang, R. Zhao, and J. Pang, "Parameter and return-value analysis of binary executables," in *31st Annual International Computer Software and Applications Conference*, ser. COMPSAC '07, vol. 1. IEEE Computer Society, 2007, pp. 501–508.

[199] L. Szekeres, M. Payerz, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. IEEE Computer Society, 2013, pp. 48–62.

[200] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[201] J. D. DeMott, "Enhancing automated fault discovery and analysis," Ph.D. dissertation, Michigan State University, 2012.

[202] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04, G. S. Avrunin and G. Rothermel, Eds. New York, NY, USA: ACM, 2004, pp. 97–107.

[203] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[204] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 497–512.