BINTYPE: A SCALABLE TYPE INFERENCE TOOL FOR COMPILED C PROGRAMS

Briti Sundar Mondal

A thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

in Information Systems Security at

Concordia University

Montréal, Québec, Canada

August 2016

© Briti Sundar Mondal, 2016

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Briti Sundar Mondal**

Entitled: **BinType: A Scalable Type Inference Tool for Compiled C Programs**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Arash Mohammadi _____ Chair

Dr. Amr Youssef _____ Examiner

Dr. Govind Gopakumar _____ External Examiner

Dr. Mohammad Mannan _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 2016 _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

BinType: A Scalable Type Inference Tool for Compiled C Programs

Briti Sundar Mondal

Reverse-engineering program binaries often relies on the recovery of high-level data abstractions. In particular, recovering variables and their type is challenging as most such information is lost during compilation. Although past proposals seem to have addressed this problem, their approaches are either not scalable and suffer from coverage issues (e.g., dynamic analysis), or yield insufficient precision by staying too conservative (e.g., static analysis). Furthermore, most recent works lift assembly to Intermediate Representation (IR), which standardizes low-level operations, and may lose some useful semantics for type inference. In this thesis, we propose BinType, a static analysis-based, scalable, precise and conservative tool that works directly on x86 assembly to automatically reveal type information of variables and function arguments. BinType is 45% more precise than TIE (NDSS'11) on a dataset 3.5 times larger, and orders of magnitude faster than its underlying algorithm. We also show that our tool makes a significant impact on the accuracy of a recent tool on binary to source matching.

Acknowledgments

I am grateful to Dr. Mohammad Mannan, Associate Professor of Concordia Institute for Information Systems Engineering (CIISE), for supervising my research work. His inspiration, encouragement and continuous support to conduct my research and to prepare this thesis is gratefully acknowledged.

I am profoundly grateful to have benefited from the thorough review of Dr. Mohammad Mannan and Xavier de Carné de Carnavalet during the writing of the thesis. I am also thankful to Dr. Mohammad Mannan for providing me all the necessary disassembled files throughout my work using IDA Pro.

Majority of this work was done as part of the National Defence NSERC Research Partnership Program in collaboration with Google project titled "Software Fingerprinting for Automated Malicious Code Analysis", in which I would like to thank specially Dr. Mourad Debbabi, Principal Investigator (PI), who recommended me the path of type inference and provided valuable feedback throughout the project. I am also grateful to the project's Co-PIs Dr. Amr M. Youssef, Dr. Lingyu Wang, Dr. Benjamin C. M. Fung and Dr. Mohammad Mannan for their guidance.

I would also like to thank Suryadipta Majumdar, Lianying Zhao and Dr. Anup Sinha for their insightful suggestions and advice, along with my lab mates for their enthusiastic discussions.

I am extremely thankful to my family member for their unconditional affection and continuous support. I dedicate this thesis to the soul of my beloved father, who

always encouraged me to touch my dream.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Compilation of high-level languages to binary inherently obscures data abstractions such as function arguments, local variables and their types, as their use is converted to a limited set of operations in registers and segmented memory regions. Recovering such lost abstractions is a prerequisite for several reverse-engineering applications, including binary program analysis [16], binary rewriting [25, 57], binary to source matching [47, 46, 39], function prototype identification [18], open-source software license violations [29] and function clone detection [36, 27].

The first step in reverse engineering data abstractions is often to identify function boundaries, then identify local variables, function arguments and returns. Identification alone has been the subject of various work, cf. [10, 11, 18, 25, 41], often relying on algorithms similar to the value-set analysis (VSA [9]). The next step is to infer the type of the identified variables, generally performed through the use of dynamic and static analysis techniques.

Dynamic analysis-based solutions utilize emulation or instrumentation tools such as: a) QEMU [13], an emulator used in Laika [21]; b) PIN [44], which controls the

execution of a process through the Linux `ptrace` API, used in REWARDS [42] and TIE [41]; or c) KLEE [19], generating inputs through symbolic execution to cover a high number of possible execution paths, used in Howard [53]. These solutions execute a program, possibly several times on various inputs, and collect and analyze runtime traces. While this process reveals actual behaviors and memory values, it is essentially resource-intensive and slow; cf. Howard [53]: "[i]t takes several hours to obtain reasonable coverage of real-life applications". To avoid long running times, only one execution trace can be analyzed at the expense of limited code coverage, e.g., REWARDS only covers about 30% of all functions and heap-allocated variables in the ten binaries analyzed in a single run.

Static analysis-based solutions (e.g., Hex-Rays [31], TIE [41], and [25]) analyze the whole program binary, since they are not limited to a given execution path, and they usually perform significantly faster than dynamic analysis approaches. However, they require additional effort to identify variables and understand memory-related behaviors that are not directly observable.

Both types of analysis share several key ideas, including type sinks, type-revealing instructions, and refining type precision as the analysis proceeds. TIE follows a combination of static and dynamic approaches to infer types, and claims better accuracy than REWARDS (for the dynamic analysis part) and Hex-Rays (for the static analysis part). TIE's authors also define the concept of conservativeness, i.e., the algorithm should prefer to output a general type (e.g., *reg32*) than a possibly incorrect precise type (e.g., *unsigned int*). They show that Hex-Rays [31], an industrial state-of-the-art reverse engineering tool that uses proprietary heuristics to infer types, often outputs incorrect precise types such as *int* instead of pointer types [41].

Most approaches also leverage Intermediate Representation (IR) such as BIL

(e.g., [41]) or LLVM IR (e.g., [25]) to standardize low-level operations, and perform their analysis on such a sanitized form. While working on IR may enable platform-independence, this may sometimes mandate the use of platform-specific details back into IR—e.g., understand the meaning of specific registers (e.g., ebp, esp) and their relationship with function arguments that are dependent on calling conventions (e.g., [41]).

Although several academic solutions have been proposed in the last decade for type recovery, and claimed better results than the leading commercial solution Hex-Rays, none provide convincing evidence to satisfy both accuracy and scalability. ElWazeer et al. [25] bring scalability to type inference with similar results in terms of accuracy as TIE (considered the state-of-the-art in this thesis), but much faster than TIE; however, their focus is on binary rewriting, which does not always require precise type recognition. Furthermore, leading proposals such as REWARDS and TIE are still unavailable to the public or the research community, several years after their publication;[1] whereas the algorithm used in ElWazeer et al. is protected by US Patent [12]. Moreover, the comparison between existing proposals is fragile at times, as only bar charts and a few selected numbers are documented. We highlight the need for making such tools publicly available, and publishing more precise results for an easy comparison.[2]

We propose a static analysis-based reverse engineering approach called `BinType`, which directly analyzes x86 instructions to automatically reveal type information of variables and function arguments from stripped binaries. We will make open-source

[1] We emailed authors of some of these tools; we either got no response, or the authors simply stated that their tool has not been maintained.

[2] For example, TIE shows the average distance between inferred and actual types for individual binary in Coreutils in a graph, but does not provide real numbers (except for 5 out of 87 binaries) and the overall score across all binaries; we extrapolate the overall score to be around 1.6 from Fig. 12 in the TIE paper [41] (the average appears to be around 1.42 in TIE's presentation slides [40, slide 49 (left)]). However, ElWazeer et al. [25] report this overall average distance to be 2 for TIE (lower is better).

our BinType implementation, and our evaluation shows that BinType is both scalable and precise, while remaining conservative as in TIE. To the best of our knowledge, only Hex-Rays works directly on disassembled instructions as we do, while others either work on runtime traces or IR. However, Hex-Rays is a closed source tool, without much documentation of its design. Moreover, in Section 5.2, we show that type inference and function arguments recovery of Hex-Rays is quite inaccurate. While several proposals also consider complex data structures (e.g., [9, 11]), we focus on the recovery of precise primitive C types, while segregating complex types, i.e., we detect when a type is complex but do not infer its precise type. Detecting accurate type for complex data structures is largely an open problem, and most current proposals are still rudimentary, see e.g., [53, 42].

We leverage IDA Pro to disassemble the target binary program, which additionally provides the function boundaries, and information on variables for each identified function. Then, our arguments recovery module extracts function arguments for each function. Finally, the assembly instructions along with variables and arguments are used for type inference. The type inference module performs static analysis to infer data types from the given context and instruction semantics (e.g., standard library function calls, and type revealing instructions), as considered in past work [42, 41]. We improve related techniques to infer types from x86 instructions, and also introduce format string analysis to improve our precision (not considered earlier). Inferred types are propagated backward and forward for typing dependent variables. At the end of the analysis, we perform an inter-procedural analysis to refine types by treating all identified function prototypes as a sink point.

1.2 Thesis statement

In this work, our main goal is to type inference from x86 assembly instructions. To achieve this objective we explore following research questions:

Question 1. Is it possible to infer variable types directly from x86 dissembled instructions in a scaleable and precise manner?

Question 2. Can we apply our extracted information to any real world reverse engineering application?

1.3 Contributions

Our main contributions are as follows:

1. We present BinType, a static analysis tool to recover the type of variables (e.g., function arguments, return, local variables) directly from disassembled x86 binaries. BinType is fast and scales well; it can process about 25,000 assembly instructions/second in a regular desktop machine. It also outperforms existing solutions in terms of preciseness and conservativeness.

2. We build on existing techniques from past work, and introduce new mechanisms, such as the use of x86 calling conventions, uninitialized registers, and format string analysis. Our result indicates that the new techniques play a significant role in identifying precise types.

3. We evaluate BinType using the most comprehensive dataset so far (3.5x larger than TIE). We use 148 binary files (704,658 assembly instructions) from six open source projects, in contrast to TIE's evaluation on Coreutils (87 binaries, totaling 203,997 instructions). Our average distance (0.87) from the source type

is better than TIE (1.6); note that, distance zero is ideal. Our inferred types are also conservative more than 91% of the time.

4. We demonstrate applicability of BinType for binary to source matching, where our extracted features (e.g., function prototypes, string literals) make a significant impact on the accuracy of an existing proposal [50].

1.4 Outline

This thesis is organized as follows. Chapter 2 describes the background and related work on type inference. Chapter 3 gives an overview of BinType. Section 4 details our design including improvements on existing techniques. After providing the experiment setup and accuracy of our obtained results in Chapter 5, we investigate a real-world application of BinType in Chapter 6 and finally, conclude our work in Chapter 7.

Chapter 2

Background and Related Work

In this chapter, we discuss the relevant background and literature related to our dissertation.

2.1 Background

2.1.1 High-level source code

High-level programming languages (e.g., C, C++, Java, Python) are used to implement computer applications, which are later transformed to software artifacts understood by a computer. All high-level source code contain human understandable syntax and semantic that allow to implement a logic in a much easier and less error prone way than low-level languages (e.g., assembly). Moreover, these high-level languages follow closer syntax between each other that allows a programmer to easily switch from one language to another. Here, we discuss some high-level features (e.g., variables (function arguments and local variables), variable types, standard library functions and formate strings) of C language, which are related to our work, using a sample example in Listing 2.1.

The sample function *read_string_from_file* receives a file name. Inside the function body, it reads a fix length of string from the file and displays it to a standard output. At the end, it returns the read status; the status value is *one* for a successful read, otherwise *zero*.

```
1 #include<stdio.h>
2 #include<string.h>
3 #include<stdlib.h>
4
5 int read_string_from_file (char* file)
6 {
7    char str[1000];
8    FILE *filePtr;
9
10   if ((filePtr=fopen(file,"r")) == NULL){
11       printf("Error! opening file");
12       return 0;
13   }
14
15   fscanf(filePtr,"%s",str);
16   printf("Value of string is: %s",str);
17   fclose(filePtr);
18
19   return 1;
20 }
21
22 int main (int argc, char* argv[])
23 {
24  int status = read_string_from_file("file.txt");
25  return 0;
26 }
```

Listing 2.1: An example of high level source code C

The function contains two local variables *str*, *filePtr* and one argument *file*; *str* and *file* are *char**, and *filePtr* is a *FILE**. It invokes four other functions: *fopen*, *printf*, *fscanf* and *flcose*, that are not defined in the program context. In C, these functions are known as standard library functions, which follow fixed specifications and their interface is defined in C header files (e.g., *stdio.h*, *stdlib.h*). For instance, the function *fopen* is used to read a file content. It receives a file name and access mode (e.g., *r* indicates only a file read) as arguments, and returns a file pointer. The function *fclose* takes the return file pointer by *fopen* as an argument and closes the file. Functions *fscanf* and *printf* are used to read data from the file and show it on the display. Both functions take a format string as arguments; format specifiers in a format string represent the arrangement of data read and write. Here, the format specifier $\%s$ indicates that a string value is read and displayed.

2.1.2 Source code compilation steps

The high-level programming language is converted to machine understandable program through the compilation, which allows a program to run on a machine. This process varies from language to language. For instance, compilation of Java source code is entirely different from the compilation of a C source code. C is run as a standalone program in x86 machine, where Java is operated on a virtual machine called Java Virtual Machine (JVM) [43]. Here, we discuss the compilation steps of C that is one of the most common programming languages [55]. The steps of C source code compilation are illustrated in Figure 1.

Compiler. A compiler transfers C source code to assembly. During the compilation, compiler removes all the high-level information (i.e., function prototypes, variables, data structures) and translates to assembly instructions. Our goal is to facilitate the reverse engineering process by retrieving this information. In Figure 2 depicts

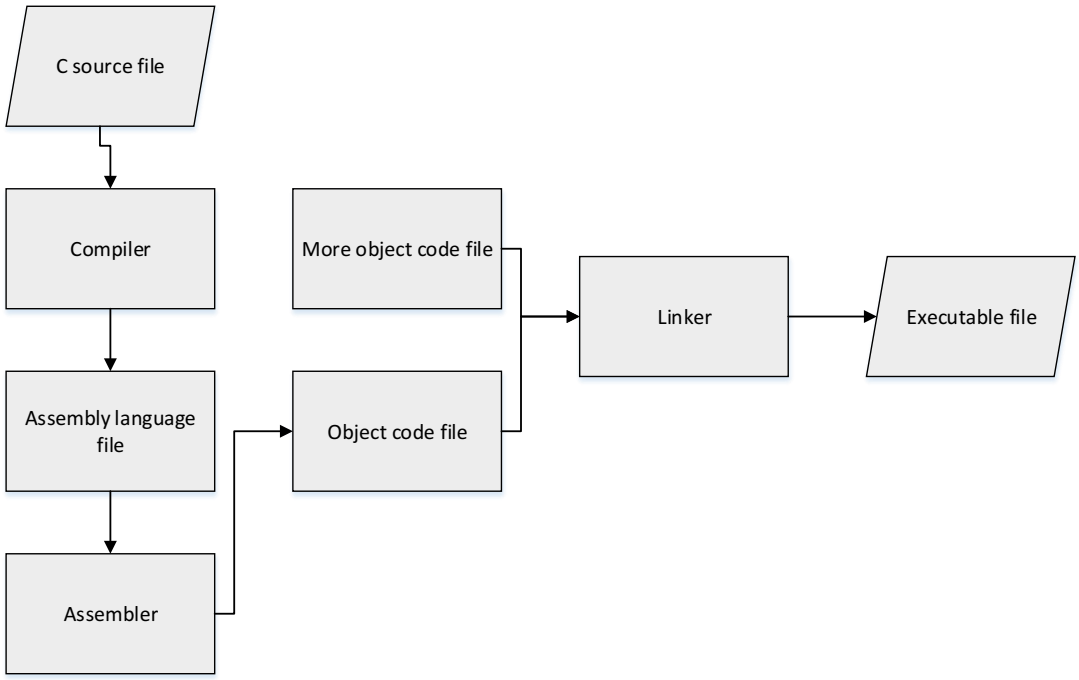

Figure 1: Steps in a compilation process

the details of a compiler's intermediate steps. We use an example source code (see Listing 2.2) taken from [40] to demonstrate the steps.

The source code contains a function *foo* that has two arguments (*char* buf* and *unsigned int* out*), a local variable *unsigned int c*, and it returns a *unsigned int* value. During the compilation, the compiler first checks the type information of received arguments, local variables and return; replaces the type information according to its data length; and they are put into the stack slots. Listing 2.3 and Listing 2.4 represent the code after replacing the type information and afterwards positions in stack slots respectively.

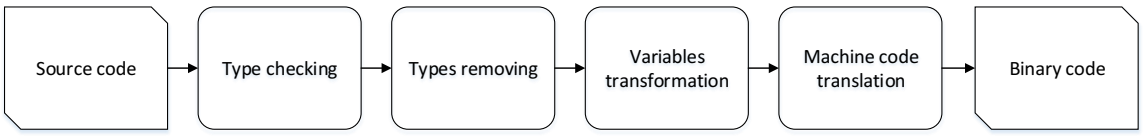

Figure 2: A typical compilation process

```
1 unsigned int foo(char *buf, unsigned int *out)
2 {
3         unsigned int c;
4         c = 0;
5
6         if (buf) {
7                 *out = strlen(buf);
8                 c = *out - 1;
9         }
10
11        return c;
12 }
```

Listing 2.2: Example source code

```
1 [32-bit] foo([32-bit] buf, [32-bit] out)
2 {
3         [32-bit] c;
4         c = 0;
5
6         if (buf) {
7         *out = strlen(buf);
8         c = *out - 1;
9         }
10
11        return c;
12 }
```

Listing 2.3: Type checking and removing

Figure 3: Function stack

Figure 3 shows the orientation of a function stack frame to demonstrate how arguments and variables are organized inside the stack. Arguments are always inserted into the top stack during the function call, where all local variables are placed into lower stack addresses. For instance, arguments buf and out are put into the top stack positioning in [+8] and [+12], where local variable c is put in lower stack [−12] in Listing 2.4. Finally, the entire code is transformed to assembly instructions where variables locations (arguments, local variables and return) are used through the direct or indirect memory accesses. At Line 4 in Listing 2.5, a value zero is assigned into the stack slot [−12] that actually holds the variable c. If we look at Line 4 of the original source code (Listing 2.2), c is equal to 0 ($c = 0$), which is equivalent as the above-mentioned assembly instruction.

```
1 [32-bit] foo(
2          [32-bit] [+8],
3          [32-bit] [+12])
```

```
4           {
5                   [32-bit] [-12];
6                   [-12] = 0;
7                   if ([+8]) {
8                   *[+12] = strlen([+8]);
9                   [-12] = *[+12] - 1;
10          }
11          return [-12];
12  }
```

Listing 2.4: Assigning variables into memory

```
1           push %ebp
2           mov %esp,%ebp
3           sub 0x28,%esp
4           movl 0x0,-0xc(%ebp)
5           cmpl 0x0,0x8(%ebp)
6           je 804844d <foo+0x2e>
7           mov 0x8(%ebp),%eax
8           mov %eax,(%esp)
9           call 804831c <strlenplt>
10          mov 0xc(%ebp),%edx
11          mov %eax,(%edx)
12          mov 0xc(%ebp),%eax
13          mov (%eax),%eax
14          sub 0x1,%eax
15          mov %eax,-0xc(%ebp)
16          mov -0xc(%ebp),%eax
17          leave
18          ret
```

Listing 2.5: Translated assembly code

Assembler and linker. An assembler translates the obtained assembly instructions into an object file. Finally, the linker creates an executable by linking the object files and required modules to run the object files.

2.1.3 Applications of type inference

In this section, we describe the usefulness of different applications that need or directly benefit from type inference.

1. **Binary to source matching** – Source code of software always allows a program analyst to understand the logic in a much easier way. In most source code of commercial software and malware is not available. Understanding the functionalities of such software needs low-level instruction analysis. Low-level programs contain less program semantics and require additional steps (e.g., extraction of control flow graphs, call graphs) to get program structures. While, identification of reused functions inside program binaries provide more details information about the software and speed up the reverse engineering process([47], [39]).

2. **Reverse engineering** – The goal of reverse engineering is to understand the binary for further analysis. The analyst may not have access the source code or debug symbols of the binary. Type inference allows to infer high-level semantics of a program's functionality by recovering the missing debug symbol tables from stripped binaries ([32], [28], [34]).

3. **Decompilation** – Decompilation retrieves a pseudo source code from assembly that provides meaningful information to understand the program ([26], [49], [20], [15]). Identifying variable types is one the most challenging tasks during decompilation [23].

4. **Binary code rewriting** – Rewriting a binary program allows a programmer to add new features, modify the binary functionality [38], enhance the security [8] and port the program into another architecture [51], [52], [33]. Typing variables accurately makes the programmer's job easier by identifying absolute memory addresses that may need to re-adjust during program rewrite.

5. **Binary code reuse** – Reusing binary code is useful when the source code is not available for that binary. Extraction of a functional code block (e.g., function) allows a programmer to add that functionality directly to many other applications. It also provides a unique benefit in malware analysis ([37], [56], [59]). One of the main challenges here is to identify the code interface that involves the recovery of function prototypes and return.

6. **Vulnerability detection, analysis, and prevention** – Another use case of type inference is vulnerability analysis [53]. Type information provides the location of variables and return addresses (e.g., pointer, buffer) and their size that enables detecting buffer overflows. The modification of this vulnerable location prevents buffer overflow exploitation.

7. **Program clone detection** – Type information can enable to identify a program. This feature improves the accuracy of a binary clone detection that is commonly used in the malware analysis ([27], [14], [24], [35]).

2.2 Related Work

In this section, we discuss work from previous research that is related to type inference. We divide the relevant approaches into four categories: (1) decompilation, (2) variables recovery, (3) static and dynamic methods and (4) arguments recovery.

We present TIE and REWADRS in more details in Section 2.3 to provide a better understanding of type inference by static and dynamic analysis.

2.2.1 Decompilation

Researchers have previously suggested the idea of using type inference to support decompilation [48], [45], [23]. However, the work done until now used to infer C types directly, while we go for a type system that allows to choose exact or closest type for each variable. This helps to avoid misleading about type information in the binaries. Moreover, some high-level features including register arguments, floating point stack variables and data types are not detected by them either. Boomerang [26], a well-known open source decompiler, has several limitations: (1) one has to assign the register arguments manually, (2) it does not detect any floating point stack operations and (3) makes a type guess (i.e., *int*) for each variable.

Phoenix is considered as an academic state-of-the-art decompiler. It is developed on top of BAP [16] and uses a modified version of TIE [41] to infer types. It improves the structure type analysis of TIE, but the analysis time makes it impractical for regular use [25].

VSA is to subdivide the memory regions into variable-like entities, based on how memory is accessed. For every memory location and register, it derives an over-estimation of the set of addresses on which the variables span, and represent them with an upper and lower bound, and a stride,

2.2.2 Variable recovery

Balakrishnan et al. [9], [10], [11] propose a binary analysis technique called value-set analysis (VSA) that could help detect syntactic structures of variables, such as field offsets, sizes, and simple structures. VSA subdivides the memory regions into

variable-like entities, based on how memory is accessed and later derives an over-estimation of the set of addresses on which the variables span. The method they use involves the binary level points-to analysis and abstract interpretation. This method allows to identifying data structures and variables from binary, not their type. Moreover, their solution is also not scalable and possibly impractical for real world implementation. Comparatively, the technique we propose to identify complex type is rather scalable and straightforward.

2.2.3 Dynamic analysis

A dynamic-analysis-based solution utilize emulation tools (e.g., PIN [44]) to execute programs, observe execution, collect runtime traces, and finally perform trace analysis to identify program variables and data structures.

REWARDS [42] proposed an idea to infer the type information from the well-known functions (e.g., standard library call, system call). Whenever a standard library function call (e.g., *strcat*) is detected inside a program, REWARDS labels the related memory locations according to the known arguments type. Moreover, it propagates the information backward and forward to refine more variables. Slowinska et al. [53] implement Howard on the top of REWARDS. They advanced REWARDS by identifying data structures on both heap and stack and complicated structures (e.g., array, nested array). However, dynamic analysis requires emulating the whole program, which is a resource-consuming and slow process. In addition, it suffers from the well-known coverage issue (i.e., not all paths are explored), which makes the analysis incapable of handling program control flows.

2.2.4 Static analysis

A static analysis-based approaches initially employ some knowledge about well-known function prototypes and instruction mnemonics. Then, its use a heuristics method (e.g.,Hex-Rays [31], DIVINE [10]) to identify the variable information and apply the employed knowledge base for type inference.

Hex-Rays [31] which is always considered as an industrial state-of-the-art tool in reverse engineering, uses proprietary heuristics to infer the type. We found out that it always guess a signed integer type for an identified variable and next try to refine the result if possible. Assigning a predetermined type without justification produces inaccurate result; the same observation is reported by TIE [41].

TIE follows a combination of static and dynamic (similar to REWARDS) approach to infer the type information, which claims better accuracy by comparing their result with REWARDS and Hex-Rays. In static analysis, TIE leverages BAP to collect the BIL instructions from binary and performs VSA to recover variable information (i.e., structures, pointers, arrays). The recovered intermediate language is suitable for platform independent solution and human understanding but does not capture the complete functionality of the input executable. High-level features such as the abstract stack, and symbols are always missing in their intermediate representations. Moreover, Slowinska et al. [53] state that VSA based approaches are less accurate to recover data structures. It cannot handle some common programming cases (e.g., *memcpy*, *alloca*), which provides a wrong estimation of variables stride length. El-Wazeer et al. [25] also shows that the TIE's approach results in a long analysis time, which makes it less practical for large executables. In addition, TIE can not handle float type that is a design limitation of BAP.

ElWazeer et al. [25] proposed a more scalable static approach than TIE [41] to identify structures from binaries. They implemented a modified version of VSA to

make their pointer analysis faster. However, they claim a good accurcy to detect pointers, nested pointers and complex data structures, but they did not detail their type inference work. Their work was more focused to binary rewrite than type inference, though they claim almost similar accuracy and conservativeness of TIE.

2.2.5 Argument recovery

Existing type inference approaches ([41], [25], [17]) do not discuss much about arguments detection. TIE mentions that it only considers memory assesses above *ebp* as arguments. This stack analysis is not adequate to find all the arguments, since arguments are also passed through registers. ElWazeer et al. [25] propose a heuristic algorithm on their lifted intermediate language to identify the register arguments. Caballero et al. [18] propose another approach to identify function arguments from executables using dynamic analysis. They collect a program execution trace and identify arguments by recognizing memory access locations. Both works show better accuracy but suffer from excessive false positives. Our function arguments recovery algorithm does not assure to identify all the arguments, but shows better precision than existing works.

2.3 Key ideas and assumptions

In this section, we first give an overview of REWARDS and TIE, as they are closely related to BinType, and as we borrow several techniques from them and some other past work. We also provide brief introduction to each technique, and our assumptions for BinType.

2.3.1 Existing solutions

REWARDS. REWARDS follows a purely dynamic analysis approach to infer types. It instruments a target binary using PIN to collect an execution trace, then infers types based on the arguments and return types from well-known functions (e.g., standard library calls, system calls), and a few type revealing instructions. Whenever a call to such functions (e.g., `strcat`) is detected inside a program, REWARDS labels the memory locations related to the function's arguments and return accordingly (e.g., *char**). Newly discovered types are propagated backward and forward to refine the type of additional variables. This process is based on constraint generation and resolution that happens during execution, as well as, post-execution. The goal is to tag memory regions with semantically sound types (e.g., structure that contains IP addresses) to guide the forensics analysis of memory dumps. We reuse the idea of type sinks, type revealing instructions and propagation/resolution through constraint generation. We also work directly on x86, however we do not have access to any runtime information.

TIE. TIE proposes a combination of static and dynamic analysis approaches. It first lifts the disassembled binary and execution trace to the BAP Intermediate Language (BIL) using the Binary Analysis Platform (BAP [16]). BIL registers and memory regions are assigned uniquely through a simple Single Static Assignment (SSA) algorithm. TIE then applies a modified value-set analysis called Dynamic VSA (DVSA) to detect variables. The type of these variables is further resolved through constraint generation based on type sinks and type revealing instructions, similar to REWARDS. TIE outputs several possible types to remain conservative. Finally, the precision is calculated by measuring the distance between the actual type and the most precise type that can be given to a variable. We follow their way of evaluating conservativeness and precision.

2.3.2 Key ideas

Type sinks. A type sink is a point in a program's instructions, where the type of one or more variables can be resolved directly. It is usually due to calls to functions that are well-defined as they follow fixed specifications. System and standard libraries functions are examples of such functions. Also, a middle size program written in C/C++ can contain a range from 1,000 to 2,500 standard library functions [30], hence such calls are a cheap and reliable source of types to start with. For example, the `strcat` function contains two *char** arguments, along with a *char** return type. The variables involved as calling parameters to `strcat` can be resolved as *char** without any ambiguity.

Type revealing instructions. An instruction that reveals meaningful information to resolve a type will be considered as type-revealing. For example, REWARDS lists the following x86 instructions used in byte-string operations as type-revealing: `movs/b/d/w` (moving), `stos/b/d/w` (storing), `lods/b/d/w` (loading), `cmps/b/d/w` (comparing), and `scas/b/d/w` (scanning). An instruction starting with *f* is related to the floating point operation, e.g., `fadd`, `fld`, `fbas`, and `fstp`; REWARDS also uses some of these operations. In addition, we use Streaming SIMD Extensions (SSE) floating-point instructions (e.g., `movss`, `addss` and `subss`), and pointer related operations, i.e., indirect memory accesses (e.g., `mov (%edx),%ebx` or `mov [mem],%eax`), which reveal pointer type information.

Type propagation. When the type of a variable is refined, it is propagated to other identified variables to update the constraints on their type. Generally, this method is used when a variable type is resolved. Its type is then recursively propagated backward (as it influences the resolution of other variables that were previously related), and forward (all future variables with constraints that depend on that variable).

Constraints generation and solving. A constraint generation derives possible

constraints based on a variable's usage. More specifically, each instruction found to affect that variable will result in a new constraint that depends on the nature of the instruction. For example, for a signed division or multiplication, TIE adds a constraint that limits the possible types to signed ones. For each instruction, constraints are generated (if any), and a solver that contains all previous program contexts tries to resolve types for any variable used by such instruction.

Value set analysis. Balakrishnan et al. [10] first propose the value set analysis (VSA) method to identify variables from executables. The goal of VSA is to subdivide the memory regions into variable-like entities, based on how memory is accessed. For every memory location and register, it derives an over-estimation of the set of addresses on which the variables span, and represent them with an upper and lower bound, and a stride, i.e., $s[lb, ub]$.

Type lattice. TIE introduces a type lattice that connects all supported types. The upper and lower bound of this lattice corresponds respectively to no type ($\top$), and inconsistent type ($\bot$). It originally assigns the entire lattice as possible types for a given variable, and later refines this range by upgrading the lower bound and downgrading the upper bound. Finally, the conservativeness and distance from the actual type is calculated. If the actual type falls between the inferred upper and lower bounds, it is considered as a conservative type; and the difference from the lower bound to actual type in source shows the distance. A lower distance indicates higher accuracy.

2.3.3 Assumptions

For now, we only consider instructions for the x86 32-bit architecture. However, with some effort, it is possible to upgrade it for the x86 64-bit architecture, i.e., additional registers and instructions should be included, some adaptation is needed to support

the calling conventions and stack formats of x86-64. Unlike TIE, BinType currently is architecture dependent. (From our experience, the use of IR to achieve platform independence interferes with accuracy.) We support only C code compiled with `gcc` and `msvc` compilers. We are agnostic to binary optimizations, but we require deobfuscated binaries. Deobfuscation is still an open problem, although some proposals are promising (e.g., [58]). In a binary, we only consider user functions (and ignore other functions such as compiler-generated ones). Finally, unused variables in source always remain untyped after our type inference, and we do not consider it as a mis-detection.

Chapter 3

Overview and Example

This chapter provides a high-level view of BinType, composed of three basic steps: assembly lifting, function prototype recovery and type inference, along with an example. An illustration of the architecture of BinType is provided in Figure 4.

3.1 Overview

Step 1: Assembly lifting. BinType lifts assembly instructions from a binary program by leveraging IDA Pro [32], a commercial disassembly toolkit. Apart from assembly lifting, IDA Pro additionally provides: (1) function boundaries, (2) invocation of library functions, identified using the Fast Library Identification and Recognition

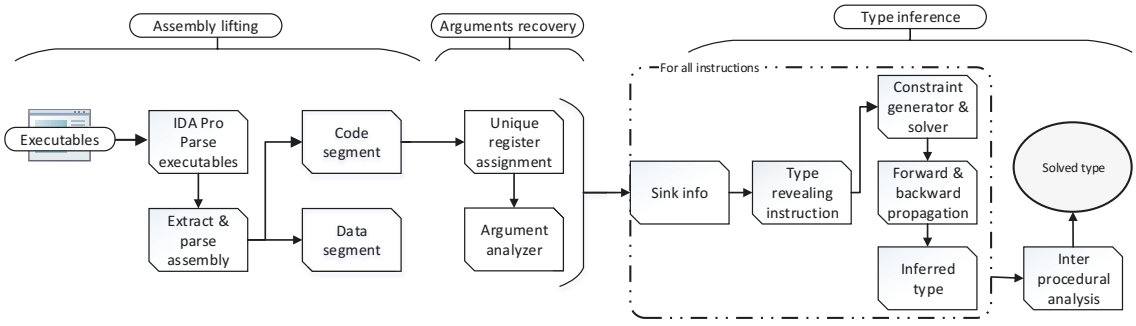

Figure 4: An overview of BinType showing different components and their work flows

Technology (FLIRT [30]), and (3) known memory addresses and offsets (e.g., string information in `_rdata`). Furthermore, IDA Pro identifies a set of variable-like entities, apparently, based on stack analysis. We divide obtained assembly instructions into a code segment (instructions from `_text` segment) and a data segment (instructions from `_data`, `_bss`, `_rodata`). In the code segment, BinType manipulates instances of registers incremented when their value is modified. For example, BinType instantiates `ecx` to ecx_1, ecx_2, etc., whenever `ecx` is updated with a new value.

Step 2: Function arguments recovery. We use the code segment obtained from Step 1 to recover function arguments. Identification of the exact number of arguments for a function highly depends on the accurate detection of the calling conventions and register arguments. We leverage Hex-Rays Decompiler [31] (later we refer it as Hex-Rays), a widely used industrial tool for decompilation, to capture the calling convention of each function. However, we do not use any information related to the number of arguments it detects, nor any other type-related information (which are quite unreliable). Based on the calling convention, BinType analyzes the stack and registers to enumerate the correct number of arguments. For example, in the *stdcall* calling convention, memory accesses with addresses above `ebp` (e.g., `ebp+4` or `ebp+8`) represent stack arguments. Respectively, an argument is counted as a register argument, if it is used without initialization inside the function boundary.

Step 3: Type inference. The variables and arguments recovered in Step 2 are passed to our type reconstruction algorithm, along with the code and data segments. BinType first looks for standard library function calls inside the code segment to extract type information (treating such calls as sink points). A standard library call can provide unambiguous type information, as the syntax and semantics of these functions are well-defined and publicly known. For instance, registers used as arguments of `strcat` are resolved as *char**. We also identify instructions that reveal partial or

precise information of a variable's type. For example, the unsigned shift left instruction `shl` indicates that the destination register has an unsigned type. This instruction is commonly used to access array indices. We further generate type constraints for variables depending on the nature of instructions. For example, in the `mov` *src*, *dest* instruction, we constraint *src* and *dest* to have the same type.

Afterwards, BinType unifies types by propagating the inferred types backward and forward to update the constraints on related variables. Finally, an inter-procedural analysis is performed to refine types by treating all identified function prototypes as a sink point.

3.2 Example

To illustrate how BinType works, we use a simple program compiled by `gcc` from the source code shown in Listing 3.1. The function `foo` has two arguments and one local variable. Listings 3.2 and 3.3 show the code and data segments respectively, from the instructions disassembled by IDA Pro of function `foo` (a dotted line indicates instructions having no effect on type recovery).

```
1 int foo (char* str, int id)
2 {
3   char buf[20];
4   if (id)
5   {
6     sprintf(buf, "%s rcv from id %d\n",id,str);
7   }
8
9   return 1;
10 }
```

Listing 3.1: Example C source code

```
1  foo  proc near
2
3  var_2C = dword    ptr -2Ch
4  s      = byte     ptr -20h
5  arg_0  = dword    ptr  8
6  arg_4  = dword    ptr  0Ch
7
8       push  ebp
9       mov   ebp, esp
10      sub   esp, 48h
11      mov   eax, [ebp+arg_0]
12      mov   [ebp+var_2C], eax
13      ...
14      cmp   [ebp+arg_4], 0
15      jz    short loc_80484C3
16      mov   eax_1, offset format
17      mov   edx, [ebp+arg_4]
18      mov   [esp+0Ch], edx
19      mov   edx_1, [ebp+var_2C]
20      mov   [esp+8], edx_1
21      mov   [esp+4], eax_1
22      lea   eax_2, [ebp+s]
23      mov   [esp], eax_2
24      call  _sprintf
25
26 loc_80484C3:
27      mov   eax_3, 1
28      ...
29      retn
```

```
30 foo  endp
```

Listing 3.2: BinType code segment parsed from IDA Pro assembly instructions for the gcc-compiled source code given in Listing 3.1

```
1 _rodata          segment dword public 'CONST' use32
2                  assume cs:_rodata
3 format           db '%s from id %d',0Ah,0
4 ...
5 _rodata          ends
```

Listing 3.3: BinType data segment corresponding to Listing 3.2

```
1 signed int __cdecl foo(int a1, int a2)
2 {
3   signed int result;
4   char s;
5
6   ...
7   if (a2)
8     sprintf(&s, "%s from id %d\n", a1, a2);
9   result = 1;
10  ...
11  return result;
12 }
```

Listing 3.4: Output of Hex-Rays Decompiler for the compiled source

Function prototype recovery. BinType detects the *cdecl* calling convention for the function `foo` from the output of Hex-Rays at Line 1 in Listing 3.4. Variables `arg_0` and `arg_4` in Lines 5-6 in Listing 3.2 are resolved as stack arguments according to

the *cdecl* convention. The code segment does not contain any uninitialized register, which indicates there is zero register arguments. Since the register `eax_3` is assigned a value before `foo` returns, the return type is not void.

Type reconstruction. Lines 3-6 (Listing 3.2) show all variables detected by IDA Pro. We attempt to identify the type information for these variables from the rest of the instructions.

Our algorithm detects a call to the `sprintf` standard library function (a sink point) with `eax_2` and `eax_1` as passing arguments at Line 24. Before the function call, parameters are initialized in Lines 21-24. From `sprintf`, both registers are resolved to *char** and *format string*, respectively. Through a backward resolution, variable `s` is resolved as *char**, and `offset format` as *format string* (Line 16).

BinType extracts the formatted string value of the `format` parameter from the `_rodata` section at Line 3 in Listing 3.3, and identifies `%s` and `%d` as format string specifiers. From Lines 17-20, BinType detects that `edx` and `edx_1` are initialized, and passed as parameters for `format`. As `%s` and `%d` represent values of types *char** and *int* respectively, `edx` is resolved as *int* and `edx_1` as *char**. Through a backward propagation of both registers, `arg_4` is resolved as *int*, `var_2C` as *char**, and `arg_0` as *char**. An integer value is copied in the return register `eax_3` at Line 27, implying that the return type is *int*.

In the end, BinType resolves all data types, including function arguments, local variables and return. Table 1 gives a summary of all variables and their inferred type, and the instructions involved in the resolution. We also show in Table 2 that for the same example, TIE and REWARDS fail to detect half of variable types. The obtained results for TIE and REWARDS are based on our analytical understanding of their algorithms, since these tools are publicly unavailable.

Table 1: Inferred types from IDA Pro assembly instructions

Variables	Inferred type	Instruction line(s)	Resolve point
`var_2C`	*int*	17, 20-21	format string (%d)
`s`	*char**	25	standard library call
`arg_0`	*int*	11-12	unification
`arg_4`	*char**	17-19	format string (%s)

Table 2: Inferred types by prior work (Hex-Rays, REWARDS and TIE) and BinType for Listing 3.2

Variables	Hex-Rays	REWARDS	TIE	BinType
`var_2C`	*int*	32-bit data	*reg32*	*int*
`s`	*char*	*char**	*char**	*char**
`arg_0`	*int*	32-bit data	*reg32*	*int*
`arg_4`	*int*	32-bit data	*reg32*	*char**

Chapter 4

Design

In this chapter, we explain the detailed methodology and the type inference algorithm behind BinType. We discuss our typing system, present mechanisms for function arguments recovery and complex type segregation, detail how type sinks, type revealing instructions and format strings are used in BinType.

4.1 BinType's type inference methodology

4.1.1 BinType typing system

Figure 5 shows the type lattice used in BinType, which is designed for the 32-bit x86 architecture. Table 3 shows how our type lattice is connected to C types in a 32-bit architecture. An unknown variable will be typed as `reg32`, `reg16`, and `reg8`, respectively for a 4-byte, a 2-byte, and a single byte value; otherwise, it will be typed as `reg(n)`.

Compared to TIE [41], our lattice additionally includes floating-point types, and adds an extra edge between complex types (`reg(n)`) and unknown types ($\top$). A pointer arithmetic or indirect memory access updates a `reg32` type variable to `reg(n)`.

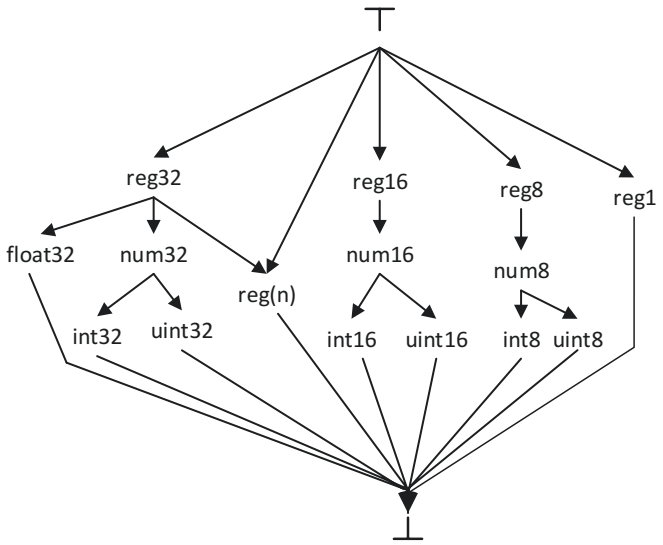

Figure 5: BinType type lattice

Table 3: Mapping between C types and types in our lattice system

C types	Types in our lattice
int *unsigned int*	*int32* *uint32*
short *unsigned short*	*int16* *uint16*
char *unsigned char*	*int8* *uint8*
** (pointer), [] (array), structure*	*reg(n)*
void	$\perp$

Additionally, a variable can also be typed as `reg(n)` directly from the unknown type ($\top$), if it contains multiple 4-byte values. We exclude `code_t` type from our type system (considered in the TIE lattice to designate the destination of *goto* instructions).

4.1.2 Function argument recovery

Identifying the correct function prototype depends on detecting the complete and accurate set of function arguments and return. If some arguments are not detected accurately, then the resulting signature of the function misleads the subsequent reverse engineering analysis. In the x86 architecture, arguments do not only pass through

the stack but also registers; however, register arguments identification remains less explored by prior work. ElWazeer et al. [25] propose a brute force algorithm for determining register arguments and return; if a register inside a function boundary is used without being initialized, it is counted as a register argument. However, they do not consider the calling conventions, and assume all registers as uninitialized at the beginning of their analysis. We show in Section 5 that their method produces many spurious arguments.

Knowing the calling convention reveals the registers and stack locations used to pass the function parameters. We follow three commonly used calling convention rules in C on x86: *cdecl*, *stdcall*, *fastcall*. In the *cdecl* and *stdcall*, memory accesses with the address above `ebp` (e.g., `ebp+4` or `ebp+8`) represent an argument. On the other hand, the *fastcall* convention adds the following constraint: the first three arguments with primitive types are passed in `eax`, `ecx` and `edx`. For all three conventions, floating point values are passed as parameters in the stack, and results are passed in the floating point stack; the register `eax` is used to return all other primitive data types.

We extract the calling convention for each function from the generated pseudo C code by Hex-Rays. Based on the convention, we analyze the stack and register arguments from the IDA Pro assembly. Memory accesses on top of the stack are always counted as arguments (common rules for passing a parameter). To obtain the register arguments, we only consider uninitialized registers that are relevant in related conventions. For unknown calling conventions, we perform an uninitialized register analysis on all registers for the first basic block. In our experiments, we noticed that most passing register arguments are moved to different memory locations at the beginning of a function start. We use this heuristic to reduce false positives. For identifying the return type, we rely on the variable that performs the last write into `eax` or the floating point stack.

```
1  foo2 proc near
2
3  var_10  = dword     ptr -10h
4  var_C   = dword     ptr -0Ch
5  var_5   = byte ptr -5
6  arg_0   = word ptr  8
7  arg_4   = dword     ptr  0Ch
8  arg_8   = dword     ptr  10h
9  arg_C   = qword     ptr  14h
10
11      push    ebp
12      mov     ebp, esp
13      sub     esp, 18h
14      push    ebx
15      mov     [ebp+var_5], dl
16      mov     edx, ecx
17      ...
18      pop     ebp
19      retn
20 foo2 endp
21 -------------------------------------------------------
22 *Identified function prototype by
23 Hex-Rays: [arguments: 4], [return: yes]
24 BinType : [arguments: 6], [return: yes]
```

Listing 4.1: Function prototype identified by Hex-Rays and BinType for the given assembly.

To demonstrate the relevance of this approach, we compare Hex-Rays with our method to detect function arguments on a simple function `foo2`; see Listing 4.1. This

Table 4: Typing rules for constraint generation and solving ($n \in 32, 16, 8$). The method *SetType* is used to update the type of *src* and *dest*. By $/_s$, we denote any signed operation, e.g., signed multiplication/division. | represents a logical OR.

Mnemonic type	Constraint	Solver
arithmetic op ∈{ +, -, *, /, %}	$\tau_{src} = num\_n \mid int\_n \mid float$ $\tau_{dest} = num\_n \mid int\_n \mid float$	if $\tau_{src} = empty$ then $SetType(num\_n, num\_n)$ else $SetTypeDest(\tau_{src})$
bitwise op ∈ { ^, &, «, »}	$\tau_{src} = num\_n \mid int\_n$ $\tau_{dest} = num\_n \mid int\_n$	if $\tau_{src} = empty$ then $SetType(num\_n, num\_n)$ else $SetTypeDest(\tau_{src})$
signed op ∈ { $/_s$ }	$\tau_{dest} = int\_n \mid uint\_n$	if $op = /_s$ then $SetTypeDest(int\_n)$ else $SetTypeDest(uint\_n)$
assignment op ∈ { = }	$\tau_{dest} = \tau_{src}$	$SetTypeDest(\tau_{src})$
unary op ∈ { ++, - -, ¬}	$dest = int\_n \mid uint\_n$	if $dest \in array\ index$ then $SetTypeVar(uint\_n)$ else $SetTypeVar(int\_n)$
typecast (*dest*, *src*)	$n = n * 2 \mid n/2$	if sizeof(*dest*) > sizeof(*src*) then $SetTypeDest(\tau_{src\_n*2})$ else $SetTypeDest(\tau_{src\_n/2})$

function takes six arguments, however, Hex-Rays can only identify four arguments (two register arguments and two stack arguments). We consider the $fastcall$ convention, which helps to detect two arguments (`dl` and `ecx`) as register arguments, and four arguments (`arg_0` to `arg_C`) from stack analysis.

4.1.3 Complex type segregation

Distinguishing complex variables is one of the most challenging part in reverse engineering. Without proper address tracking, it is difficult to recover complete information. We only differentiate complex types (structures, arrays, pointers) from primitive types, a process we refer to as *complex type segregation*. For this purpose, we follow VSA [9]: we track indirect memory accesses in *[base + index × scale + disp]*, where *base* and *index* are registers, and *scale* (1, 2, 4, or 8) and *disp* (displacement, e.g., any 32 bit value) are constants. We consider each memory region (the upper and lower bounds) as a one-bit array with value all zeros. During the analysis, we perform the

address arithmetic and change a value from zero to one inside a memory boundary, based on whether it is accessed or not. MEM $\{1_1,1_2,1_3,...1_i,....0_{n-1},0_n\}$ implies that the first i bytes of an n-byte memory block are accessed by the program. If we detect a memory access to a variable that is not inside its known boundaries, we update the variable length by adding the amount of size it accesses. For example, if the address of a variable with four-byte size is moved into a register *eax* (e.g., *mov eax, [var]*), and the next instruction tries to access a value from *[eax + 4]* (e.g., *add ecx, [eax+4]*), we infer that the variable holds a pointer, and the subsequent instruction merely accesses the next four bytes. In the end, we obtain the length of all variables and their memory layouts. A variable with more than a four-byte length is counted as a complex variable. The identified memory layouts can be used to detect more specific information about memory regions.

4.1.4 Format string analysis

C provides a set of standard library functions for formatting strings. For example, *scanf* and *printf* are used to read and write formatted data, and *error*, *error_at_line* are applied for formatted error logs. A format specifier (beginning with %) is used to generate this format string. An argument is passed to replace the respective specifier with the argument value in the resulting string. The prototype of a format specifier [54] is defined by *%[flags][width][.precision][length]specifier*. The types of arguments inside the format string is identified from the format specifier characters (e.g., $\%d$ implies signed integer).

We create a list of standard library functions taking format strings as an argument. We identify a pattern (e.g., *mov [esp], offset format; call printf*) of represented format string in a program binary, and then the type information from it. We iterate through each character of the format string for a % (format specifier) that indicates the starting

point of formatted data. We then identify the corresponding specifier character, the argument corresponding to the specifier, and finally, assign a type for the argument based on the specifier character.

4.1.5 Constraint generation and solving

In our work, constraint generation and solving is mainly concerned with resolving primitive type variables. Complex types are already identified through our complex type segregation (see Section 4.1.3), type revealing instructions and sink points. We apply the rules described in Table 4 for constraint generation and solving.

If a variable involved in a binary operation is identified as a pointer, that operation is considered as a pointer arithmetic operation, and the destination variable must be a pointer as well. Similarly, binary arithmetic instructions involve integer or float point computations. Thus, each operand can be an integer, number or float type. If the source operand does not have a type then the solver assigns the number type (num_n) for both the source and destination variables. Otherwise, the destination variable will be the same type as the source. Similarly, the use of unary operators (increment and decrement) on an array index allow us to infer that the type of the operands is (at least) unsigned. Assignment instructions indicate that the source and destination have similar types.

4.1.6 Type sinks

BinType identifies the invocation of a standard library function call to determine the type of a variable. BinType uses IDA FLIRT [30] to identify standard library functions from program instructions. FLIRT extracts a pattern of each standard library function and stores it in a signature database. A match with this signature during the assembly analysis will be treated as a library function call. IDA Pro 6.8

contains 30MB of signatures for x86, which is almost 100 times larger than the type sinks considered in TIE and REWARDS.

4.1.7 Type-revealing instructions

BinType considers all the type revealing instructions from REWADRS. In addition, we consider instructions that provide partial information about a type, and take into account the SIMD Extensions (SSE) floating-point instructions (e.g., *addss*, *subss*).

4.1.8 Signed and unsigned types

Most past proposals in type inference [41, 42, 53, 25, 17] do not consider jump instructions (e.g., *jge*, *jnl*), which appear to be a good source for signed type. Deng et al. [22] first create a list of jump instructions to perform signed type analysis, which is used for static integer overflow vulnerability detection in Windows binaries. We reuse the jump instructions from Deng et al. in our work.

We divide these instructions into two sets: signed and unsigned type instructions. If a variable or register is used as an argument of a memory allocation function, or an index of an array, then it must be an unsigned type. Moreover, variables compared by different x86 conditional jump instructions, e.g., *ja*, *jae*, *jb*, *jbe*, *je*, *jne* are typed as unsigned, while variables in instructions *jge*, *jnl*, *jng*, *jnge* are typed as signed.

4.2 Type inference algorithm

Algorithm 1 outlines the steps in BinType's type inference. Given a binary program, our algorithm resolves all variable types including the arguments and return. The inputs are the code and data segments, along with all identified variables.

```
Notation used (more in the description below):
  I_i: i^th instruction from asm_file;
  tr_ins: List of type revealing instructions;
  sinkdb: List of sink points;
  m_i: Instruction mnemonic for i^th instruction;
  C: Constraints for m_i;
  Γ: List of constraint operators;
  IDAVar: List of variables from Hex-Rays;
  Var: List of all identified variables and their types;
  Var_op: Similar variable list for the operand op;
Input: Disassembled binary file asm_file
Output: Identified variables and types (Var)

Function resolve_type(asm_file)
    foreach I_i ∈ asm_file do
        if I_i = mov(src, dest) then
            update_dependency_list(src, dest);
        end
        else if m_i ∈ tr_ins then
            τ = get_type_from_instruction(m_i);
            unify_type(op, τ, m_i);
        end
        else if I_i ∈ sinkdb then
            τ = get_sink_type_from_signature(m_i);
            unify_type(op, τ, sp);
        end
        else if m_i ∈ Γ then
            C = generate_constraint(m_i);
            τ = solve_constraint(C, Γ);
            unify_type(op, τ, cs);
        end
    end
    return Var;
end

Function update_dependency_list(src, dest)
    Var_dest.append(src);
    Var_src.append(dest);
    if src ∈ IDAVar then
        τ = get_type(src);
        if (τ) then
            unify_type(src, dest, τ, uf);
        end
    end
end

Function unify_type(op, τ, type_src)
    foreach var ∈ Var_op do
        if type_src of var > type_src then
            var ← τ;
        end
        unify_type(var, τ, type_src);
    end
end
```

ALGORITHM 1: BinType's type inference algorithm

The *resolve_type* method. This is the main method of our type inference algorithm. For each instruction I_i in an assembly file, the method checks whether I_i is: (1) an *assignment* instruction (e.g., *mov*, *lodsx*, *lea*, *stosx*), (2) a type revealing

instruction, or (3) a type sink. Additionally, the method checks the possibility of generating constraints from the instruction mnemonic. The details of these checks are provided below.

1) Assignment instruction (Line 15). An *assignment* instruction implies that the destination (*dest*) has a similar type dependency as the source (*src*). This constraint is added in both variables' dependency list by triggering the *update_dependency_list* method with *dest* and *src* as the arguments.

2) Type revealing instruction (Line 18). If the instruction mnemonic is an entity of type revealing instructions, then the *get_type_from_instruction* method is called to extract the type from the instruction mnemonic m_i. The calling method holds the type information for the type revealing instructions, and retursn the corresponding type of the received argument. The identified type (τ) is then propagated to all variables that should have the same type by calling the *unify_type* method.

3) Sink point (Line 22). If a function call in the instruction I_i is a type sink, then the prototype of that function is obtained from the *get_sink_type_info_from_signature* method. This method contains all the standard library functions, along with prototypes that are identified by IDA FLIRT for the target executables. The calling method finds the received function name into the list, and returns the matching function prototype. Variables that are used as arguments of the function call, will be typed according to the function prototype. Finally, unification (the *unify_type* method) is applied to propagate the type information to all the function contexts.

4) Constraint generation and solving (Line 26). If an instruction mnemonic (m_i) is found in the constraint generation list in Table 4, then the *generate_constraint* method derives possible constraints (C) based on the way a variable is being used, and calls the *solve_constraint* method to retrieve the type. The *solve_constraint* method applies the solving rules (Γ) (at column three, in Table 4), using the current

execution context, and returns a suitable type (τ) for that constraint variable. A solver does not guarantee that it will always return a type for every constraint, but if the type is resolved, then the unification process (*unify_type*) is applied to propagate the type information.

After completing the analysis for each instruction, a list of variables and their types is returned by the *resolve_type* method.

The *update_dependency_list* method. This method receives the source (*src*) and destination (*dest*) as arguments from the caller function, and adds them to each other's dependency list. A dependency list of a variable contains the variables and registers that have a similar data structure as that variable. Here, the dependency list of *dest* (Var_{dest}) adds *src* as a relevant type variable, and vice-versa. If *src* is an entity of the variable list (i.e., variables identified by IDA Pro), then type information for that variable is requested through the *get_type* method. A valid type (τ) is then passed through both *src* and *dest* dependency lists using the *unify_type* method.

The *unify_type* method. Whenever a type is resolved, this method is called to propagate the type information. A back propagation is recursively applied to propagate the identified type to all variables that should have the same type. Following arguments are received by *unify_type*: operand name (*op*), type (τ), and type source (*type_source*). *type_source* holds the value from where a type is resolved: (i) sink point (*sp*), (ii) type revealing instruction (*tri*), (iii) constraint and solver (*cs*), and (iv) unification (uf). We prioritized the source points as follows: $sp > tri > cs > uf$. The type identified from a higher priority source is disseminated during the back propagation. Thus, our approach helps to traverse deeper in the type lattice, which results more accurate type. For example, a variable *var* is recognized as $num32$ through the constraint solver. Later, a sink point returns $int32$ for that variable. A sink point always provides the exact type information, therefore we put it in the top

priority. According to the priority order, our method modifies the *var* type $num32$ to $int32$, which allows the *var* to go one step down in the lattice. The *unify_type* method first extracts each variable (*var*) from the operand dependency lists (Var_{op}); then it replaces the old type with the new one, if the priority of the current type source *var* is less than the inferred type source; and finally, it calls the *update_type* method again using that *var* as an operand to solve its dependent variable list. This procedure is continued until all variables from the list is updated.

Example. We illustrate step by step execution of our algorithm with a simple example in Table 5. The first column of the table presents assembly instructions, where I_i denotes i^{th} instruction in an assembly file; the second column provides the variable Var with its dependency list as well as their type; third and last columns provide how type information is inferred from different source points, and how a type is resolved through backward propagation. Our algorithm finds a match with a *mov* instruction mnemonic in I_1 and I_2, where the *src* operand is moved to a 32 bit register eax, and then moved to the stack. For the *mov* instruction, the algorithm updates the variable *src*'s dependency list Var_{src}, by adding the destination register eax_0; also *src* is added in Var_{eax_0}; in a similar way, Var_{eax_1} and Var_{dest} are updated by *dest* and eax_1, and typed as $reg32$. The *strcpy* invocations in instruction I_5 is a type sink that implies st_0 and st_4 must be of type $char*$. Every new type identification always ends up with a backward resolution by calling the *unify_type* method. The third column shows how the backward propagation works, and updates the type information of a variable. The method takes the dependency list of Var_{st_0} and Var_{st_4}, and types its variables eax_1 and eax_0 as $char*$; next, it takes the dependency list of eax_1 and eax_0 and resolves the type. Thus, all variables of Var_{eax_1} and Var_{eax_0} is resolved by backward propagation, and variables *src* and *dest* are typed as $char*$.

Table 5: Running example for the type inference algorithm. A type with following *sp* and *bp* indicates that the type is identified either from sink point or through backward propagation

Instructions	Variables : {Similar type list} {type}	Backward propagation	Type resolution point
1. *mov eax , [ebp + src]*	src : $\{eax_0\}$ $\{reg32, char*: bp\}$ eax_0: $\{src\}$ $\{reg32, char*: bp\}$	(bp: τ_{eax_0}) : $\tau_{src} = char*$	*mov* instruction
2. *mov [esp + 4], eax*	eax_0 : $\{src, st_4\}$ $\{reg32, char*: bp\}$ st_4: $\{eax_0\}$ $\{reg32, char*: sp\}$	(bp: τ_{st_4}) : $\tau_{eax_0} = char*$	*mov* instruction
3. *lea eax, [ebp + dest]*	$dest$: $\{eax_1\}$ $\{reg32, char*: bp\}$ eax_1: $\{dest\}$ $\{reg32, char*: bp\}$	(bp: τ_{eax_1}) : $\tau_{dest} = char*$	*mov* instruction
4. *mov [esp], eax*	eax_1: $\{dest, st_0\}$ $\{reg32, char*: bp\}$ st_0: $\{eax_1\}$ $\{reg32, char*: sp\}$	(bp: τ_{st_0}) : $\tau_{eax_1} = char*$	*mov* instruction
5. *call strcpy*		τ_{st_0}: $char*$ τ_{st_4}: $char*$	Sink point: *char* strcpy(char *dest, const char* src)*

4.2.1 Inter procedural type analysis

As a last step of the algorithm, we run an inter procedural analysis. We get the type of all function arguments, local variables and the return for each function from the previous step. We then use this information to refine the typing information one more time. Variables that are still untyped has a high chance to be typed during this inter procedural analysis. We create a function list using these identified function prototypes. We generate a call graph using IDA Pro for the entire program and perform a depth first search (DFS) from the root node (i.e., *main*). The edge between two nodes implies a function call, and nodes are treated as a caller and callee. The variables of the caller will be typed using the caller as a sink point (similar to a standard library function call).

Chapter 5

Evaluation

5.1 Implementation and experiment settings

BinType is implemented on top of IDA Pro. We write 4.5K SLOC C# code for parsing the assembly instructions, recovering the function prototypes and identifying

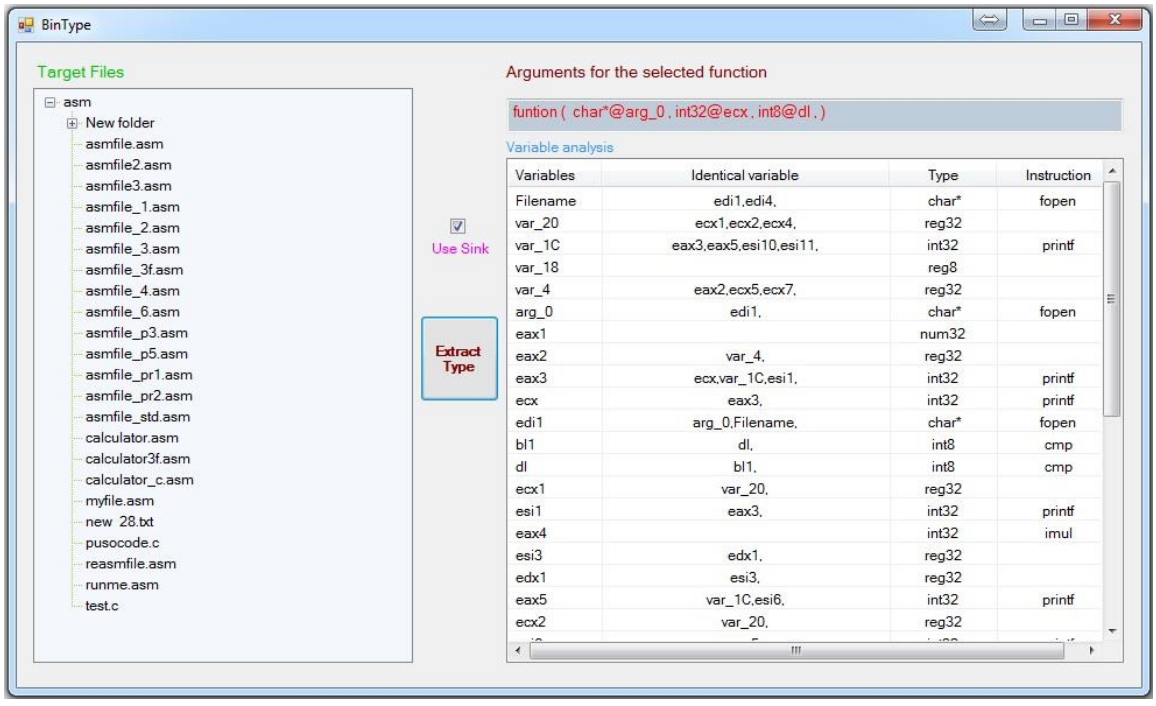

Figure 6: User interface of BinType

type information. The user interface of BinType is showed in Figure 6.

We take 148 binaries (704,658 instructions) from six commonly used C open source projects: procps-3.2.6 [6] (with 19.1K SLOC), iputils-20020927 [2] (with 10.8K SLOC), net-tools-1.60 [5] (with 16.8K SLOC), coreutils-5.93 [1] (with 117.5K SLOC), sqlite [7] (with 119.7K SLOC) and miniz [4] (with 3.7K SLOC). We conduct the following experiment on a machine with a 3.10 GHz Core i7-3770S processor and 8GB RAM running Windows 7 64-bit operating system.

For comparison, we need the actual variable types from the source code. Instead of parsing the source (which may be unreliable due to function inlining), we first compile all source programs in debug mode, and then use libdwarf [3], a C library, to obtain DWARF information (with variable types) from ELF object files. Libdwarf is an essential tool used in type inference ([41], [42]) for reading DWARF information from application binary. We mainly use this tool to extract and organize the stack variables. Some of these variables may be scattered in the source code, which can be meticulously arranged with libdwarf. The output of libdwarf variables is used for automated comparison with the output of BinType. While BinType resolves type information for whole executables, we evaluate only the outcomes of the user-level code (e.g., exclude compiler functions). Note that we do not use debug binaries as input to BinType.

5.2 Experimental results

In this section, we show the accuracy of type inference and function arguments identification, and scalability of BinType.

Accuracy of BinType. We evaluate the accuracy of BinType by measuring the differences between the types of variables declared in the program source code and those derived from the executables by BinType.

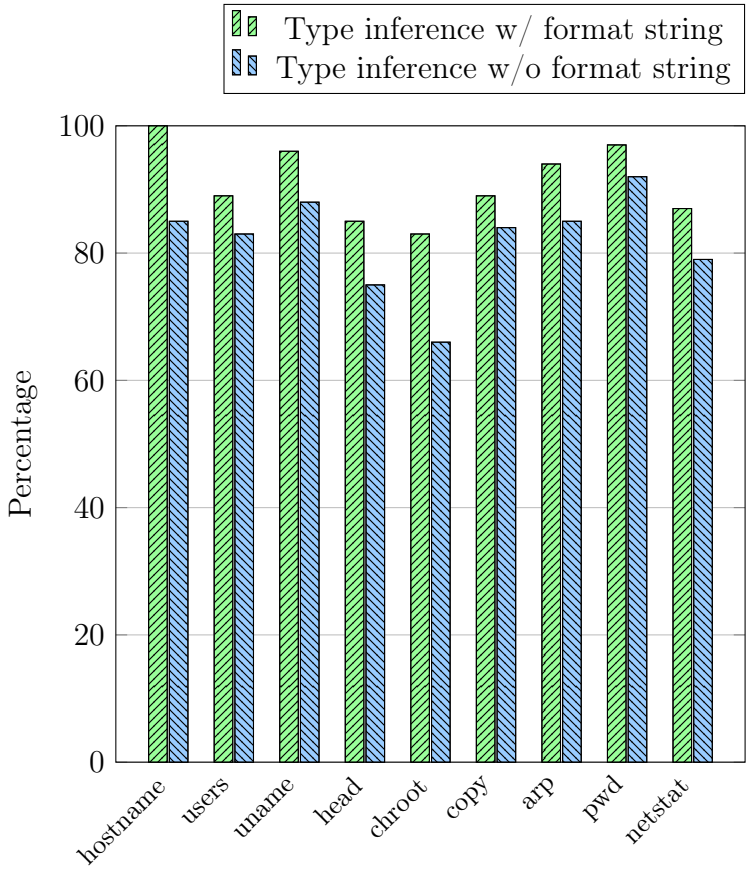

Figure 7: Impact of format string in type inference

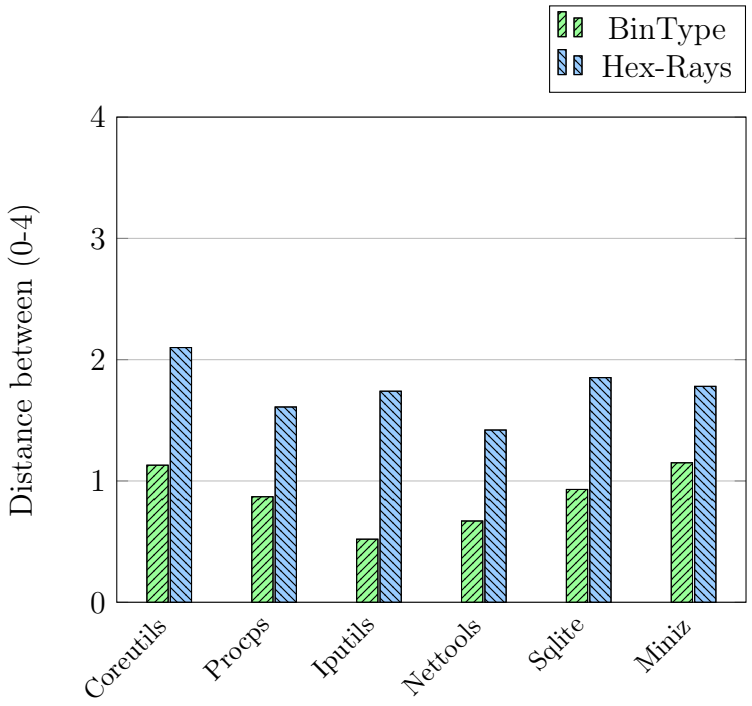

Figure 8: Distance of each variable type from the original type (lower distance indicates better accuracy)

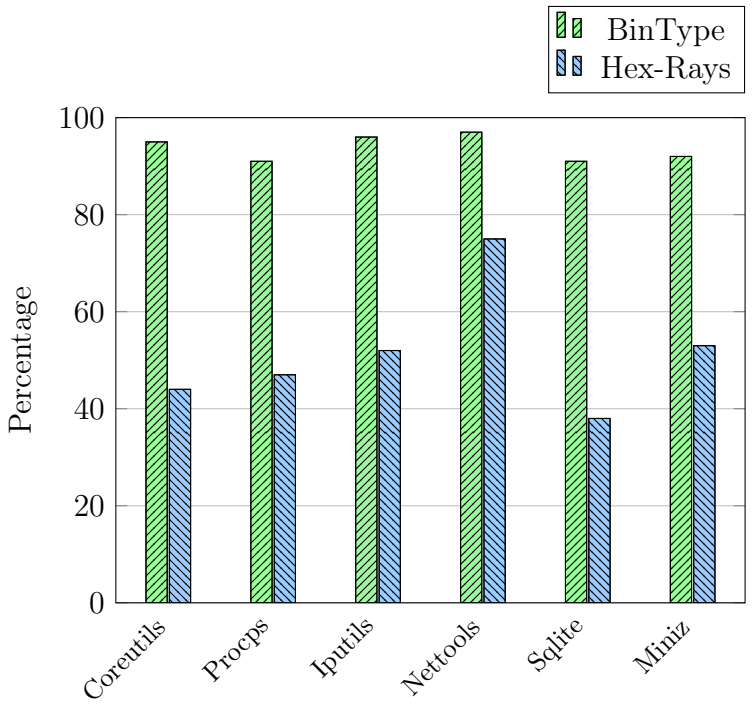

Figure 9: Conservativeness of identified variable types by BinType and Hex-Rays

Table 6: Comparison between BinType and TIE's static analysis (Table 2 in [41])

	Conservativness (ideal=1)		Distance (ideal=0)	
Program	TIE	BinType	TIE	BinType
chroot	0.87	0.82	1.76	0.72
df	0.942	1.00	1.42	0.07
groups	0.93	1.00	1.52	0.00
hostid	0.97	1.00	1.63	0.00
users	0.97	1.00	1.51	0.27
Average	0.93	0.96	1.57	0.21

First, we compare our accuracy against Hex-Rays. For both BinType and Hex-Rays, we assign all identified complex data structures to a single complex type. Our aim is to see the efficiency of both methods to identify complex variables, not to extract exact complex type, which is a challenging problem (and most current tools report results with doubtful accuracy, see e.g., [53, 42]). If the identified complex variable represents any complex type (e.g., array, pointer or structure) inside the source, we consider it as a match. Finally, we calculate the conservativeness and accuracy (distance from the exact type) of a variable using TIE's metrics. Variables with a smaller distance are more accurately typed (ideal distance is zero). Figures 9 and 8 illustrate the conservativeness and distance of variables type identified by BinType and Hex-Rays. In both cases, BinType outperforms Hex-Rays. BinType is around 32% more conservative (93% vs. 61%), and offers closer distance (0.87 vs. 1.75) than Hex-Rays.

Second, we compare our accuracy with TIE [41] using the similar metrics and binaries (87 binaries from Coreutils) used in their evaluation. In both type lattices, the maximum distance between levels is four. In our case, the upper bound of each identified variable is assigned as $reg32$, $reg16$, $reg8$, or $reg(n)$ based on their length. All inferred types at the end of the analysis are counted as lower bounds. We measure the distance by calculating the difference between the lower bound and actual type. We also calculate the conservativeness by comparing the exact type position inside

the type boundary. Our results show that BinType is 95% conservative, which is almost similar compared to TIE (the exact value is unavailable, the bar charts of TIE presentation slide [40]). Since our structural type distance calculation method does not follow the similar method in TIE, we only compare our results with TIE's real type average distance 1.42 (obtained from TIE slides, apparently better than what is reported in the paper). Our average distance (1.13) is better than TIE, indicating that BinType is a more precise type inference tool. Table 2 in TIE points to actual conservativeness and distance values of five binaries. Our comparison with their samples is given in Table 6. Note that, for TIE's dynamic analysis, conservativeness (1.0) and distance (1.15) metrics are better, but the code coverage is under 9%.

Format strings. We select nine widely used modules from our selected dataset, and compute the effectiveness of format strings for type inference. Figure 7 presents our accuracy results with and without the use of format strings; as apparent, format strings improve accuracy by almost 9%.

Arguments recovery. To verify the strength of our proposed technique in Section 4.1.2, we estimate the accuracy of identified function arguments. From our analysis, we find that BinType's argument recovery algorithm can successfully identify all stack arguments, but sometimes misses some register arguments (mostly due to the presence of very few uninitialized registers outside the first basic block). We compare our work against Hex-Rays, and get slightly better precision (3%) (see Figure 10).

We also measure the results against the function prototype recovery algorithm proposed by ElWazeer et al. [25] and Caballero et al. [18]. ElWazeer et al. use a static approach that can detect all stack and register arguments with 0.20 false positive per function. A dynamic analysis technique is used by Caballero et al., which claims the similar detection rate with a better false positive rate (0.15 false positive per function). We provide better results than both algorithms in terms of precision, 94%

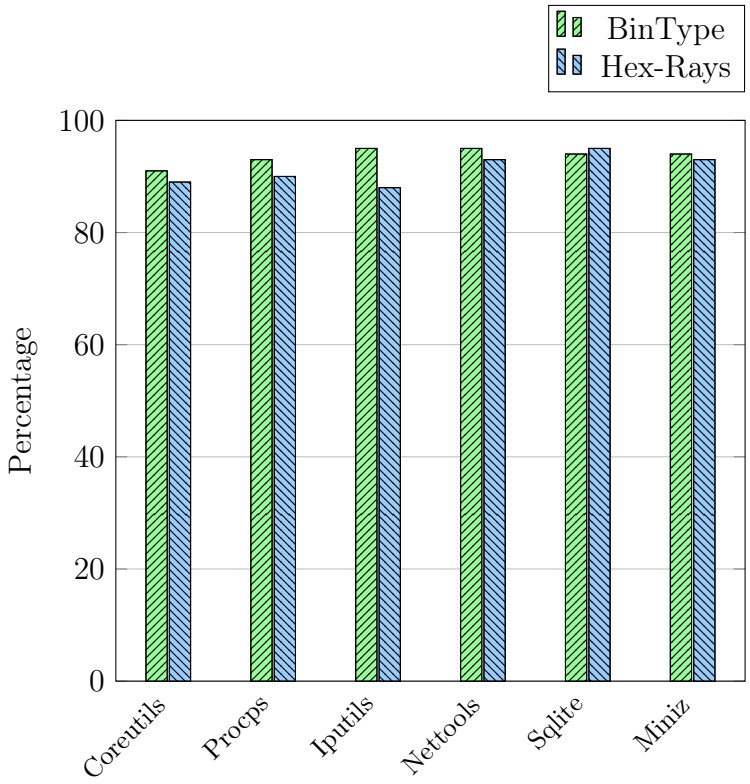

Figure 10: Precision of identifying function arguments

(BinType) vs. 83% ([25]) and 86% ([18]), and false positive rate, which is 0.06% for BinType.

Scalability. We measure the number of instructions for each project and the time to infer the type information. Table 7 shows the average time for each module. We can process about 25,000 instructions/second, compared to DIVINE [10]'s 10.8 instructions/second (averaged from Table 2 in the 2007 DIVINE paper; machine

Table 7: BinType scalability

Source base	Assembly instrunctions	Time (sec)
Coreutils	263337	12.30
Procps	15623	0.56
Iputils	51706	2.00
Nettools	19893	0.78
Sqlite	322716	13.90
Miniz	31383	1.25

configuration is unavailable). To the best of our knowledge, most prior work used DIVINE, either directly or a modified version of DIVINE's algorithm (i.e., VSA), for identifying variable types. Note that, compared to prior work, our complex variable analysis is primitive, which partly helps our scalability.

Chapter 6

Application of BinType

BinType can be applied to a number of applications. In this chapter, we demonstrate how BinType provides unique benefits to improve the binary to source similarities. We develop an application called CodeBinPlus by extending an existing proposal CodeBin [50], which is used to find the binary to source function similarities. The main challenge in identification of reused functions in a binary program is to find out common features within a binary and source code. As extended features in CodeBinPlus, we use the extracted function prototypes and string literals of BinType, and modify the score matrix corresponding to features.

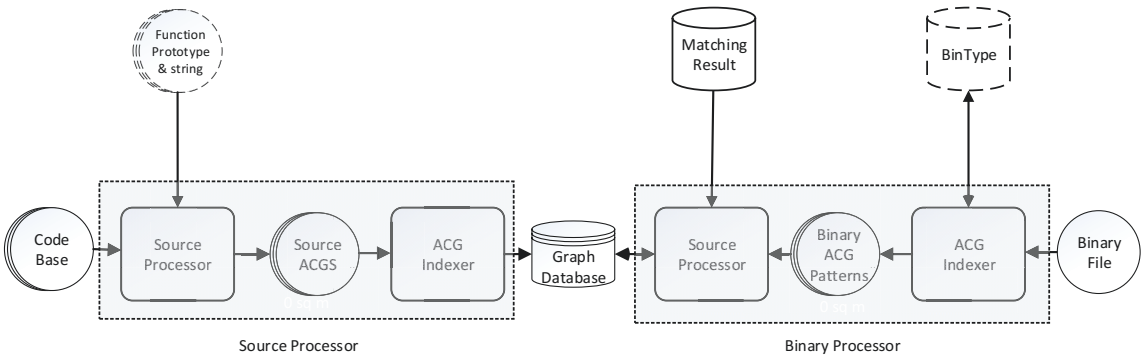

Figure 11: Overall design of CodeBinPlus (adapted from CodeBin [50])

Table 8: CodeBinPlus test dataset

Projects	Functions	Lines of Code
Linux Kernel	236,08313,	416,043
Sqlite	1,859	99,674
Coreutils	2,124	59,673
Miniz	120	5,260
Curl	64	9,434
Zmap	360	11,638
Unqlite	2,682	84,347
Tor	4,241	179,282
Git	5,880	157,858
Putty	2,275	80,882
Redis	2,952	46,407
Hashkill	2,674	420,719

6.1 CodeBinPlus

CodeBinPlus builds on top of CodeBin. Overall, our approach is designed around the following procedure (see Figure 11).

Source processor. First, CodeBin parses and analyzes the source code of different C projects, and transforms them into annotated call graphs (ACGs) by extracting specific features and relationships, such as internal functions calls, number of function arguments, complexity of control flow, and calls to standard library functions and system APIs. CodeBinPlus extends this feature list by adding function prototypes and string literals from source functions. Indexed function prototypes do not contain the exact type information for arguments. Since BinType may output types that are not defined in C (e.g., `num32`), we only use primitive vs. complex information. When ACGs are created from source code, they are stored in a database accompanied by specific indices to allow fast searching. All codebases will be stored in a single database with unique labels, allowing for lookup operations over many different codebases.

Binary processor. For each target binary file, CodeBin disassembles and analyzes the binary to extract the same features as source, creating binary ACG patterns.

CodeBin leverages Hex-Rays to lift its required features. We already discussed Hex-Rays' limitation to identify the correct number of arguments and function prototypes. CodeBinPlus uses BinType to collect this information, as well as string literals. ACG patterns extracted from binary functions are then converted into queries that can be run against the source graph database.

Score matrix. CodeBinPlus assigns a score to each extracted ACG based on the presence of distinctive features, such as library and API calls, the number of function arguments, string literals and the total number of binary functions included in the pattern. It takes the output of CodeBin, and scores it one more time with the newly added features.

Finally, we evaluate CodeBinPlus using several open source projects; we present the results of code reuse and no use detection through binary to source matching by searching 261,314 functions from 12 different C projects (see in Table 8).

6.2 Evaluation

We use two executable binaries, which reuse all or portions of previously indexed projects. These binaries are created by compiling (default settings) open source libraries. All tests are run on a Windows 7 x64 desktop machine powered by an Intel Core i7-4790 CPU with a frequency of 3.6 GHz, utilizing 8GB of memory. We run CodeBin and CodeBinPlus on our indexed database and compare the obtained results (we received access to CodeBin's source only, not their reported dataset).

Results collection and verification. We evaluate CodeBinPlus and CodeBin on miniz and sqlite binaries that reuse parts of previously indexed source code. In the results, a binary function may be correctly matched to a unique source candidate, or may be mapped to several source candidates. We have included the percentage

of cases where the correct source candidate is uniquely identified (top 1), or listed among the top 3 or 5 functions.

Reuse detection. Our evaluation results on CodeBinPlus and CodeBin show that approximately 87% of miniz and 85% of sqlite reused functions are uniquely and correctly matched to their source code, where CodeBin reports 69% and 71%, respectively. CodeBinPlus can identify 95% correct source candidates in top 3. In contrast, we found, CodeBin detects 84% correct source candidates in top 3. Figure 12 shows the effectiveness of BinType's features to refine the candidate list. Our features improve the accuracy of BinType around 20% to identify the unique result.

No reuse. We run one more test to verify the significance in a no reuse scenario. We remove miniz source code from our indexed database, and run the same test again with a binary containing miniz functions. CodeBinPlus and CodeBin still return some matches in this case. CodeBinPlus recognized 14 functions, which match at least one candidate from the indexed functions, in contrast to CodeBin's 43 (ideal case is zero).

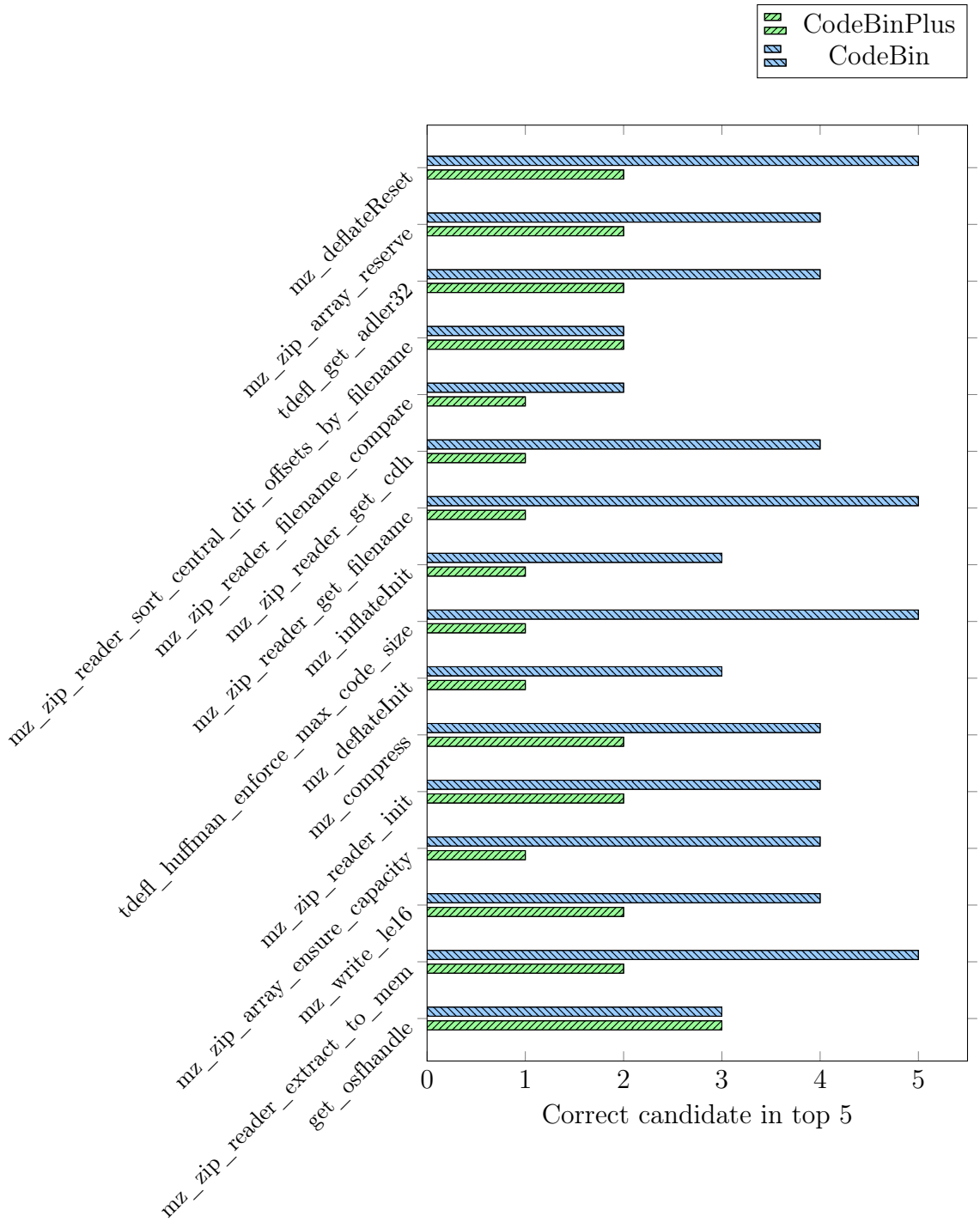

Figure 12: Efficiency of BinType features to converge the ranking results

Chapter 7

Conclusion

Reverse engineering binary programs, more specifically recovering variables and their types have many application cases such as binary to source matching, identifying function prototype, binary rewriting, function clone detection. Hence, any automatic mechanism to recover variables along with their types from binary programs is considered significantly valuable.

In this work, we presented BinType, a type inference system on x86 assembly instructions. Unlike the previous works such as [41], [42] which are not scalable and take long analysis time in large binary, we provided a scalable solution on a large dataset that was magnitude faster than the previous solutions. We improved related techniques to infer types from x86 instructions and also introduced format string analysis to improve our precision. We also improved the accuracy of function arguments recovery by introducing unique uninitialized registers identification technique. We showed that our algorithm is more precise than other existing proposals. Moreover, we applied our extracted features as an application in binary to source matching and demonstrated a good accuracy to identify reused functions.

Bibliography

[1] Coreutils: Gnu core library. Source code. `http://www.gnu.org/software/coreutils/coreutils.html`.

[2] Iputils: Linux networking library. Source code. `https://github.com/iputils/iputils`.

[3] Libdwarf. GitHub project (Feb. 2005). `http://reality.sgiweb.org/davea/dwarf.html`.

[4] Miniz: Zip archive. Source code. `https://www.openhub.net/p/miniz`.

[5] Net-tools: Linux networking tools. Source code. `https://sourceforge.net/projects/net-tools`.

[6] Procps: Command line utilities. Source code. `https://gitlab.com/procps-ng/procps`.

[7] Sqlite: Database utilities. README file. `http://www.sqlite.org/src/doc/trunk/README.md`.

[8] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[9] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction (CC'04)*, Barcelona, Spain, Mar. 2004.

[10] G. Balakrishnan and T. Reps. DIVINE: discovering variables IN executables. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, Nice, France, Jan. 2007.

[11] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS'10)*, 32(6):23, Feb. 2010.

[12] R. Barua and M. Smithson. Binary rewriting without relocation information, Aug. 13 2013. US Patent 8,510,723.

[13] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Anaheim, CA, Apr. 2005.

[14] M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.

[15] P. T. Breuer and J. P. Bowen. Decompilation: The enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1613–1647, 1994.

[16] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification (CAV'11)*, Snowbird, UT, USA, July 2011.

[17] J. Caballero, G. Grieco, M. Marron, Z. Lin, and D. Urbina. ARTISTE: Automatic generation of hybrid data structure signatures from binary code executions. Technical Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, Aug. 2012.

[18] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Network and Distributed System Security Symposium (NDSS'10)*, San Diego, California, USA, Feb. 2010.

[19] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI'08)*, San Diego, California, USA, Dec. 2008.

[20] C. Cifuentes. *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, 1994.

[21] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Operating Systems Design and Implementation (OSDI'08)*, San Diego, California, USA, Dec. 2008.

[22] Y. Deng, Y. Zhang, L. Cheng, and X. Sun. Static integer overflow vulnerability detection in windows binary. In *International Workshop on Security (IWSEC'13)*, Okinawa, Japan, Nov. 2013.

[23] E. Dolgova and A. Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105–119, 2009.

[24] T. Dullien, E. Carrera, S.-M. Eppler, and S. Porst. Automated attacker correlation for malicious code. Technical report, DTIC Document, 2010.

[25] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *Programming Language Design and Implementation (PLDI'13)*, Seattle, WA, USA, June 2013.

[26] M. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Working Conference on Reverse Engineering (WCRE'04)*, Delft, The Netherlands, Nov. 2004.

[27] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi. BinClone: Detecting code clones in malware. In *Software Security and Reliability (SERE'14)*, San Francisco, California, USA, June 2014.

[28] I. Guilfanov. Simple type system for program reengineering. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 357–361. IEEE, 2001.

[29] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 63–72. ACM, 2011.

[30] Hex-Rays. Fast library identification and recognition technology (F.L.I.R.T.). Online documentation (May 2015). `https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml`.

[31] Hex-Rays. Hex-Rays decompiler. `https://www.hex-rays.com/products/decompiler/`.

[32] Hex-Rays. IDA Pro disassembler. `https://www.hex-rays.com/products/ida/index.shtml`.

[33] R. J. Hookway and M. A. Herdeg. Digital fx! 32: Combining emulation and binary translation. *Digital Technical Journal*, 9:3–12, 1997.

[34] E. R. Jacobson, N. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM, 2011.

[35] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *Machine Learning and Applications (ICMLA)*, Boca Raton, FL, USA, Dec. 2012.

[36] 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*, San Francisco, CA, USA, May 2013.

[37] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010.

[38] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software: Practice and Experience*, 24(2):197–218, 1994.

[39] F. Leder. RE-Google plugin documentation. Readme file (2009). `https://www.hex-rays.com/contests/2009/REGoogle/README.TXT`.

[40] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs (slides). Conference slides. (Feb. 2011). `https://www.internetsociety.org/sites/default/files/lee2.pdf`.

[41] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, USA, Feb. 2011.

[42] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, USA, Feb. 2010.

[43] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education, 2014.

[44] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, June 2005.

[45] A. Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming (ESoP'99)*, Amsterdam, The Netherlands, Mar. 1999.

[46] A. Rahimian. BinSourcerer. GitHub project (Feb. 2005). `https://github.com/BinSigma/BinSourcerer`.

[47] A. Rahimian, P. Charland, S. Preda, and M. Debbabi. RESource: a framework for online matching of assembly with open source code. In *Foundations and Practice of Security (FPS'12)*, Montreal, QC, Canada, Oct. 2012.

[48] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Principles of Programming Languages (PoPL'99)*, San Antonio, TX, USA, Jan. 1999.

[49] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*, Washington, DC, USA, Aug. 2013.

[50] A. Shahkar. On matching binary to source code. Master's thesis, Concordia University, Montreal, QC, Canada, 2016. `http://spectrum.library.concordia.ca/980919/`.

[51] G. M. Silberman and K. Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. *Computer*, 26(6):39–56, 1993.

[52] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.

[53] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, USA, Feb. 2011.

[54] The Open Group. Base specifications issue 7. 2013 Edition. `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/printf.html`.

[55] TIOBE Software BV. Tiobe index. Online article. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[56] D. Urbina, Y. Gu, J. Caballero, and Z. Lin. Sigpath: A memory graph based approach for program data introspection and modification. In *European Symposium on Research in Computer Security*. Springer, 2014.

[57] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *USENIX Security Symposium*, Washington, DC, USA, Aug. 2015.

[58] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2015.

[59] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.