ADVANCED SOFTWARE REVERSE ENGINEERING

by

Zhibo LIU

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Supervised by Prof. Shuai WANG

November 2021, Hong Kong

TABLE OF CONTENTS

ADVANCED SOFTWARE REVERSE ENGINEERING

by

Zhibo LIU

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

Software reverse engineering is a practice of understanding the intentions of software developers by analyzing and extracting design and implementation information from compiled software. It works as an automatic process of converting the incomprehensible binary files into human-readable high-level language. As the cornerstone of various cybersecurity tasks, software reverse engineering widely enables malware analysis, off-the-shelf software security hardening, cross-architecture code reuse, and vulnerability detection, in which cases the source code is usually unavailable.

Because of its importance, software reverse engineering has evolved over the last few decades and has developed into a complete and systematic process. Typically, software reverse engineering on the x86 platform consists of three parts as disassembly, lifting, and decompilation. The disassembler translates hex values in executable into assembly instructions at the disassembly stage. Then the assembly instructions will be lifted into a variety of different Intermediate Representations (IRs) by the lifter at the lifting stage. Finally, at the decompilation stage, the decompiler converts IR into comprehensible high-level language.

Accurate software reverse engineering is notoriously hard because the compiler discards unnecessary information during converting source code to machine code. Nowadays, there are lots of studies focused on recovering more information from binary executables. However, the perfect solution remains do not exist. Therefore, to help understand the challenges and the state-of-the-art (SOTA) methodology of software reverse engineering techniques,

we conduct a thorough survey related to the existing researches and a detailed comparison among the different methods at different stages.

According to the procedure of software reverse engineering, we first introduce the difficulties in three stages: disassembly, lifting, and decompilation. Afterward, we discuss recent works related to disassembly and reassembleable disassembly techniques and their pros and cons. Then we compare and divide existing lifting techniques into two categories of emulation-style and succinct-style. Moreover, we also show the advanced decompilation techniques for converting IR to pseudo C code. Finally, we discuss the emerging fields and application scenarios of decompilation technology. After a thorough survey of the existing literature, we also suggest several potential future research directions that could drive the development of software reverse engineering techniques. We believe our survey will benefit the entire reverse engineering community.

CHAPTER 1

INTRODUCTION

Software reverse engineering is the process of converting incomprehensible binary software into human-readable or analysis-friendly high-level representation. In today's cybersecurity practice we will encounter lots of software without available source code, such as malware and close-sourced software. To avoid malicious damage or ensure the security properties of software, we often need to manually or automatically analyze such software. However, the lack of source code hinders the analysis and makes further fixing or hardening impossible. To fulfill the analysis of such software, the methodologies of software reverse engineering are summarized and developed.

The manifestation of software reverse engineering is converting low-level machine code into higher-level representations. As the output of software reverse engineering, the high-level representation can be either compiler IR [49] or a high-level programming language, depending on the downstream applications. Some reverse engineering platforms also designed their own IRs [5, 16]. The IRs tend to be more succinct than machine code and contain variable types and function information for analysis and understanding. Besides, generally the IR would be designed as platform-independent so that the binary executables from different platforms can be converted to a unified IR and processed with the same analysis tool, avoiding re-implementing the wheels.

Software reverse engineering serves as the paramount component and the trust base in numerous cybersecurity tasks. By applying disassembly, lifting, and decompilation techniques, software reverse engineering enables and promotes many security-critical tasks such as malware analysis, vulnerability detection, off-the-shelf software security hardening, and cross-architecture code reuse. With the increasingly complex software ecosystem and the emergence of a variety of software threats such as ransomware, lots of research is devoted to improving the accuracy and reliability of reverse engineering techniques. Therefore, to help understand the methodologies of the state-of-the-art reverse engineering techniques and their wide applications, we conduct a thorough survey related to the existing literature and a detailed comparison among the different techniques.

To be more specific, we first introduce the basic concepts, workflow, and vast applications of software reverse engineering. Following the workflow, we then present the challenges of

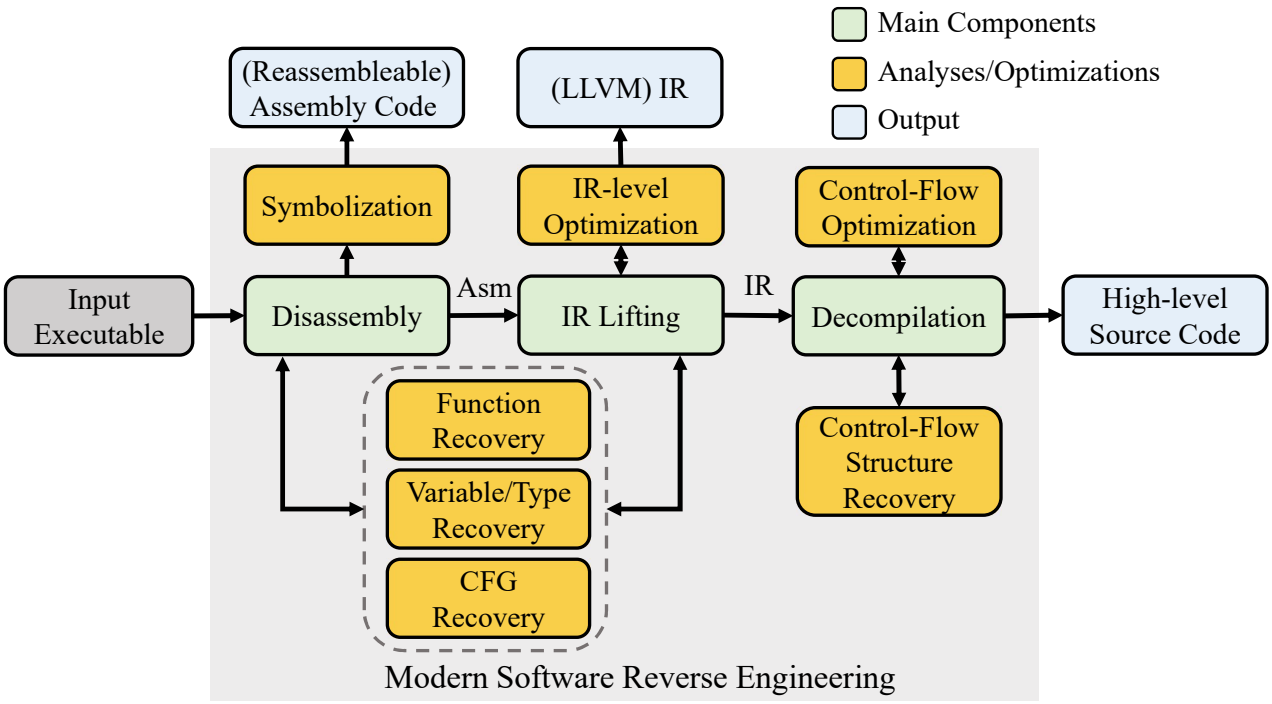


Figure 1.1. Workflow of modern software reverse engineering.

software reverse engineering with the SOTA solutions. In the end, we discuss the emerging field of decompilation research and some potential research directions for further improving the capacity of software reverse engineering.

CHAPTER 2

BACKGROUND

Before the literature review of the existing research efforts, we first present the preliminary knowledge of software reverse engineering, including the domain-specific concepts and notations. In 1990, the definition of software reverse engineering (SRE) is proposed by Institute of Electrical and Electronics Engineers (IEEE) as "the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction" in which the "subject system" is the end product of software development [21]. For the last decades, as the basis of lots of security-critical tasks, software reverse engineering techniques have been continuously explored and developed by reverse engineers and researchers. Nowadays, modern software reverse engineering has formed a three-component pipeline that consists of disassembly, lifting, and decompilation.

2.1 Workflow

We demonstrate the workflow of modern software reverse engineering in Fig. 1.1. A complete software reverse engineering process starts with a binary executable without available source code. First, the disassembler will parse and disassemble the binary file into assembly code with function boundaries recovered at the disassembly stage. Next, the disassembled assembly code, as the input of the lifter, will be lifted to (customized) IR with variables and types recovered depending on the characteristics of IR. Lastly, at the decompilation stage, the program control flow will be recovered and optimized in order to be translated into the pseudo-high-level programming language. With some advanced techniques, the output of the disassembler even could be symbolized to be able to reassemble into a functional binary executable. In this case, the assembly or IR can also be used to support downstream applications without further processing.

2.2 Concepts

In this section we detail the concepts and basics of common components in the software reverse engineering process.

2.2.1 Disassembly

Conceptually, disassembly is the process of translating machine code into assembly language, it is the inverse process of assembly [59, 67, 13]. Machine code is bytes that are read and executed by a machine. While machine code is unreadable by humans, disassembly converts machine code into a human-understandable form by converting hexadecimal values into assembly language represented as visible characters. For instance, each unique hexadecimal sequence on the left corresponds to one and only one assembly instruction, in this case, disassembly is simply about mapping the sequence to the assembly code:

```
; machine code
0x55
0x89 0xe5
0x53
0x83 0xec 0x74
```

```
; disassembly code
push ebp
mov  ebp, esp
push ebx
sub  esp, 74h
```

Linear and Recursive Disassembly. While translating a hexadecimal value sequence into an assembly instruction is straightforward, the problem is where to find the hex values to translate. Classical static disassembly techniques can be summarized as two methods. The first method, named *linear disassembly*, translates the hex values to assembly instruction sequentially from the start of the text segment. However, in modern compilers, data and code may be interleaved for optimization reasons. Thus the second method, named *recursive disassembly*, is leveraged to circumvent the interleaving problem. Recursive disassembly will translate instructions following the control flow of the program, so no data will be misinterpreted as code. Nonetheless, statically recovering control flow from binary is generally hard when indirect jumps exist (e.g., *call eax*) [61]. Thus in practice, this problem is usually solved by combining linear and recursive disassembly as *speculative disassembly* [24].

Reassembleable Disassembly. While traditional disassembly aims to readability and expressiveness only, Wang et al. first proposed the *reassembleable disassembly* which aims to produce output that can be reassembled back to a functional program [67]. By fixing the symbolization problem, reassembleable disassembly can enable lots of binary rewriting applications including software security hardening and cross-architecture code reuse. More details of the reassembleable disassembly technique and related work will be discussed in Sec. 4.1.2.

In the process of software reverse engineering, input executables will first be fed into the disassembler, while the machine code is translated into assembly instructions, the data sections within each executable will also be identified for further usage. Existing research on disassembly has been shown to work very well in practice and the proposed methods

can smoothly disassemble large-size binary executables [11, 45, 67].

2.2.2 Lifting

With a variety of available IRs, LLVM IR, as a *strong-typed* IR defined for LLVM compiler framework, is used by most reverse engineering tools. With the support of many LLVM analysis and optimization passes, LLVM IR can directly bridge reverse engineering tool development and compiler framework, avoid reimplementing the wheels by reusing existing LLVM passes, and show great potential and research value. Therefore, we focus on LLVM IR (hereinafter referred to as IR) and related lifting techniques in this survey.

The assembly code output by the disassembler will be fed to the lifter as input. The machine-specific detail is discarded during lifting, leading to a more refined platform-independent representation of the program (IR). The key procedure of lifting is translating each assembly instruction to a sequence of semantic-equivalent IR statements:

```
; assembly code
mov  dowrd ptr [rbp -4], edi
```

```
; lifted IR
%1 = add i64 %RBP, -12
%2 = load i32, i32* %EDI
%3 = inttoptr i64 %1 to i32*
store i32 %2, i32* %3
```

The lifted IR statements would faithfully emulate machine execution, including CPU register-level computations, memory updates, and other side effects. A sequence of IR statements, corresponding to one machine instruction, will be usually represented as a *translation rule* and hardcoded in lifters. Hence, each machine instruction will be mapped to a translation rule and translated into an IR sequence.

Emulation-style vs. Succinct-style. The LLVM IR is a strong-typed IR, which means all variables and their types are explicitly defined in LLVM IR and branches are dependent on predicate variables decidedly. Therefore, given the lifted IR code, the central focus is to recover variables, types, and high-level program control flows from low-level IR code. To recover program variables, some tools already proposed and implemented needed static analysis and inference techniques [4, 11, 10, 57, 29]. To recover variable types, constraint-based type inference systems are typically formulated [47, 53]. However, recovering variables and types from stripped binary is challenging and existing methods are either inefficient or inaccurate enough (we will detail the challenges and existing method at Sec. 3.4 and Sec. 4.2.2).

To avoid facing type and variable recovery challenges directly, lifter developers have made compromises by trying to recover variables and types but do not guarantee the correctness, which is denoted as *succinct-style*, or trying to emulate the assembly instruction

at hardware level thus ensuring correctness at the cost of efficiency, which is denoted as *emulation-style*. We will compare the two lifting styles in detail in Sec. 4.2.1.

2.2.3 Decompilation

> *"... machine-specific detail is discarded, leading to the essence of the program (source code), from which it is possible to divine a deeper truth (understanding of the program)."* [65]

Although IR is already understandable, the IR is designed with only limited and straightforward kinds of basic operations (load, store, arithmetic operations, etc.) for better analysis and optimizing friendliness. Limiting operations to a few simple and basic ones allow the compiler developer to dispense with complex operations considerations in an optimization strategy. However, overly simple instructions make it difficult for analysts to summarize and understand the higher-level intent of the program. Thus decompilation technique is used to further extract the essence of a program by converting it into a more advanced programming language-like form [65, 33, 30, 18].

While the high-level data-flow information, including variables and types, is recovered at the lifting stage, the control flow information in IR still retains a simple assembly-like structure:

```
; branch in assembly
 cmp     [rbp - 12], 5
 jge     label
```

```
; branch in LLVM IR
 %2 = icmp slt i32 %1, 5
 br i1 %2, label %3, label %4
```

To recover high-level control flow structures, modern decompilers implement a set of structure templates and search to determine whether an IR code region matches the predefined patterns:

```
; LLVM IR
  ...
  br label %4
4:
  ...
  %6 = icmp slt i32 %5, 5
  br i1 %6, label %7, label %13
7:
  ...
  br label %4
13:
  ...
  ret i32 0
```

```
; decompiled code
 while (x < 5) {
   ...
 }
 ...
 return 0;
```

Some advanced techniques enable an iterative refinement to polish the recovered structure. To date, techniques have been designed to guarantee the structure recovery correctness and also to improve readability [18, 71]. In addition, modern decompilers usually design optimizations to polish the lifted IR code, including dead code elimination and untiling [18, 23, 44]. Also, reverse engineering of C executable files might encounter "chicken and egg" problems (e.g., data flow analysis relies on the precise output of control flow analysis, and vice versa). Indeed, analysis and optimization modules can be invoked back and forth for iterations; the output of one module is used to promote the analysis of other modules [44].

2.3 Survey Scope

In this survey, we focus on the challenges of software reverse engineering and existing methods. Since software reverse engineering has been studied for over decades, voluminous studies on various problems in reverse engineering have been published in many domains. To make this survey intuitive and concise, we focus on landmark solutions to various problems in the field of software reverse engineering and recent research published in top-tier conference and journal of the following areas:

- **Security**: IEEE Symposium on Security and Privacy(S&P), USENIX Security Symposium, ACM Conference on Computer and Communications Security (CCS), The Network and Distributed System Security Symposium (NDSS)
- **Programming Language**: ACM SIGPLAN Conference on Programming Language Design And Implementation (PLDI), Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).
- **System**: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), USENIX Annul Technical Conference (USENIX ATC), ACM European Conference on Computer Systems (EuroSys).
- **Software Engineering**: International Conference on Software Engineering (ICSE),ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), IEEE/ACM International Conference on Automated Software Engineering (ASE) and The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).

CHAPTER 3

CHALLENGES

According to the pipeline of software reverse engineering and existing literature, several challenges inhibit the further improvement of the accuracy and reliability of reverse engineering techniques. Briefly speaking, the challenges can be summarized into one question: how to recover information lost during compilation from the striped binary? To answer this question, software reverse engineering researches can be divided into several different aspects.

3.1 Differentiating Code from Data

As we mentioned in Sec. 2.2.1, data and code can be interleaved in x86 [19]. One of the reasons is that compilers aggressively interleaves static data within text segment for better performance. Also, code is not aligned and can start from any offset of the executable segments [13], which means developers can arbitrarily interleave data and code in hand-written assembly code. The fact that the Intel x86 instruction set allows variable instruction size further aggravates the problem of code/data distinction. Although linear and recursive disassembly can be combined to solve some cases, there are unavoidable false positives and false negatives. As the lowest and most fundamental component in the software reverse engineering pipeline, disassembly must be perfectly accurate to support the downstream pipeline. Any small error in the disassembly stage can be magnified later and change the final result. While it is challenging to differentiate code from data, some researches seek to get around this problem by brute force disassembling or probabilistically disassembling [13, 52]. These approaches can guarantee the correctness of disassembly in a certain probability at the expense of performance and efficiency.

3.2 Symbolization

Another complex problem to solve in disassembly is the symbolization problem. This problem exists when we reassemble the disassembled code back to a functional program for binary rewriting. In general, there are four main steps in the automatic binary rewriting

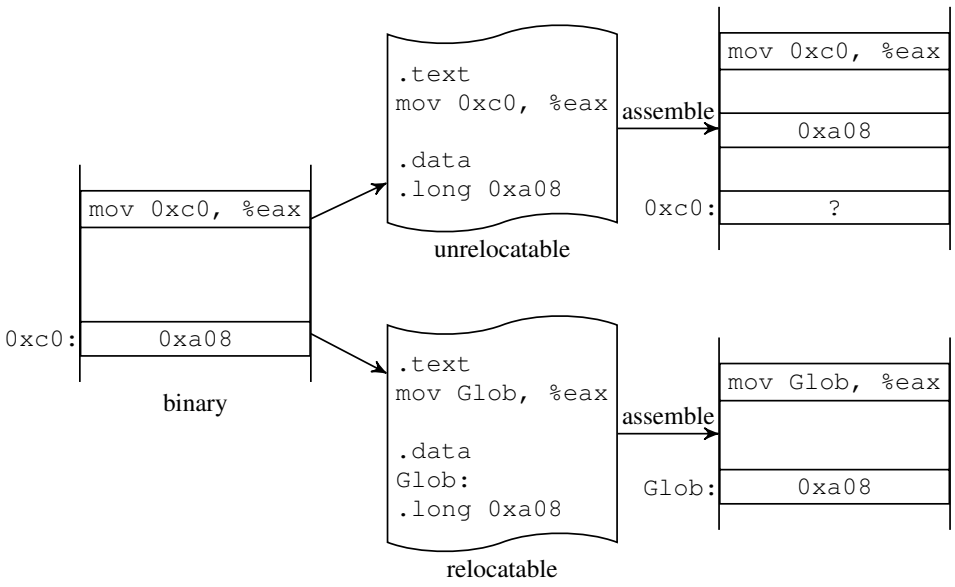


Figure 3.1. Relocatable and unrelocatable assembly code [67].

process, (1) disassembling binary into assembly code, (2) performing static analysis, (3) performing transformations, and (4) assembling code back into a usable binary executable. In most cases, program transformations will inevitably change binary layouts. However, in the machine code, a global variable will be represented as its memory address, and it is hard to distinguish it from constant values. If we take a memory address as a concrete value and keep it unchanged, the reassembled program will very likely be defective. In other words, the global variables need to be symbolized before reassembling in order to produce a functional program.

This problem is also known as the relocatable problem, as shown in Fig. 3.1. If the global variable *Glob* is not symbolized (unrelocatable) and remains the same value of 0xc0, the program will be unusable because 0xc0 points to an unknown memory region after reassembling. The relocatable problem was first proposed by Wang et al. in 2015 [67] and has sparked a series of consequent research [66, 70, 27] that we will discuss in detail in Sec. 4.1.2.

3.3 Variables Recovery

A variable in source code could be mapped to a low-level register or a memory offset after compiling to binary. The variable can be either a local variable, a global variable, or a function argument. Classifying a register of memory offset as a variable is known as the variable recovery problem. It is challenging to recover variables from stripped binary because information about variables is lost at compile time.

```
unsigned int foo(char *buf,
         unsigned int *out)
{
  unsigned int c;
  c = 0;
  if (buf) {
    *out = strlen(buf);
  }
  if (*out) {
    c = *out - 1;
  }
  return c;
}
```

(a) Source code

```
push   %ebp
mov    %esp,%ebp
sub    $0x28,%esp
movl   $0x0,-0xc(%ebp)
cmpl   $0x0,0x8(%ebp)
je     8048442 <foo+0x23>
mov    0x8(%ebp),%eax
mov    %eax,(%esp)
call   804831c <strlen@plt>
mov    0xc(%ebp),%edx
mov    %eax,(%edx)
mov    0xc(%ebp),%eax
mov    (%eax),%eax
test   %eax,%eax
je     8048456 <foo+0x37>
mov    0xc(%ebp),%eax
mov    (%eax),%eax
sub    $0x1,%eax
mov    %eax,-0xc(%ebp)
mov    -0xc(%ebp),%eax
leave
ret
```

(b) Disassembled code

Figure 3.2. Example of source code and disassembled code [47].

For example, consider the disassembled code shown in Fig. 1(b), compiled from the source code in Fig. 1(a). In the first step, variable recovery should infer that (at least) two parameters are passed and that the function has one local variable. Classical approaches recover the variable information by looking at *access patterns* [47, 48], e.g., one typical access pattern is accessing a memory block via the ebp register with offsets, with which we can infer there are two unique parameters (0xc and 0x8). However, because compiler optimizations may break such patterns, these approaches are not fully reliable.

3.4 Types Recovery

Similar to the variable recovery problem, to convert the disassembled code to high-level, *strong typed* IR, we need to recover the types of variables. We face a similar dilemma as type information is lost at compile time as well. There are only memory blocks of different sizes at the binary level, so typical methods seek to infer types from context information. One straightforward method to infer types is propagating information from executed *type sinks*, which are calls to functions with known type signatures, e.g., functions from C standard libraries [48]. However, such type sinks are not always available in binary code, some research formulated constraint-based type inference systems to recover variable types [47, 53]. These methods further improve accuracy, but are still far from being able to support type-aware analysis, e.g., pointer analysis.

3.5 Control-Flow Structure Recovery

In the decompilation stage, high-level structured control flow constructs such as loops, if-then-else, and switch will be recovered from assembly code in the Control Flow Graph (CFG) form. As discussed in Sec. 2.2.3, modern C decompilers implement a set of structure templates and are able to guarantee the structure recovery correctness and also to improve readability [18, 71]. These methods require accurate CFG to be recovered in advance. However, when indirect jumps exist in the binary code, recovering CFG needs non-trivial data-flow analysis or dynamic execution information. Also, indirect jumps may significantly reduce the readability of decompilation results. In the fact that object-oriented language (such as C++) compiled binary frequently uses indirect jumps for dynamic dispatch, some research focuses on reverse engineering of object-oriented code [40, 58, 31]. Nonetheless, due to the complexity of object-oriented code, there is no standard solution exists. Thus in this survey, we mainly focus on reverse engineering of C code.

CHAPTER 4

EXISTING SOLUTIONS

To improve the accuracy and reliability of software reverse engineering techniques, numerous methods are proposed to tackle the challenges listed in Chapter 3. Similarly, we follow the pipeline of software reverse engineering to introduce these existing solutions.

4.1 Disassembly

As discussed in Chapter 3, one of the major difficulties in disassembly is differentiating code and data. While some successful disassembly tools exist on the market [38, 44, 54, 2], these tools inevitably have substantial false positives (take data as code) and false negatives (ignore code as data). In this section, we mainly discuss several state-of-the-art disassembly and binary rewriting techniques that solved the problem to a large extent.

4.1.1 Disassembly with Less Errors

Bauman et al. proposed *superset disassembly* that employs the brute force method to guarantee no false negatives [13]. Specifically, superset disassembly takes every offset in the text segment as one possible start of an instruction, called superset instruction. It finds intended sequences of instructions by brute force disassembling every possible instruction, although the bytes of adjacent instructions may overlap. In this way, it can be guaranteed that there is no false positive in the result, and thus enabling error-free binary rewriting. However, the disassembled program will be bloated as lots of false-positive instructions exist. The bloated instructions could cause substantial size and runtime overhead because getting the instruction to be executed needs a table lookup each time, especially in practice, a binary writer based on superset disassembly has to instrument all superset instructions. Thus, Miller et al. proposed *probabilistic disassembly* based on superset disassembly to reduce the number of false positives further [52]. Probabilistic disassembly aims to reason the inherent uncertainty in the binary analysis caused by the lack of debugging information. The basic idea is to compute the probabilities of each address being the true positive start of an instruction. They define three hints that imply true positive instructions and assign prior probabilities

to these hints, then perform probabilistic inference to summarize the confidence of true positives from these evidence.

Besides the false negatives free but costly disassembly approaches, some research aims to disassembly faster while keeping a low false positives/ negatives rate. Flores-Montoya *et al*. present `Ddisasm` [32], a disassembly framework based on Datalog combining static analysis and heuristics, achieving lower false positives and false negatives rates compared with `Ramblr` [66]. This framework takes advantage of Datalog, enabling faster empirical evaluation of new heuristics and analyses. In addition, `XDA` (Xfer-learning DisAssembler) leverages transfer learning to use different contextual dependencies learned from machine code for accurate disassembly [56]. Although this framework surpasses SOTA disassemblers on both speed and accuracy, errors in results cannot be easily fixed without retraining, which prevents machine learning-based frameworks from being iteratively updated frequently. Also, it requires expensive GPUs to outpace existing tools on speed.

4.1.2 Symbolization

Even though we can get correct disassembly code, there is still a huge gap between achieving successful binary rewriting. This gap is summarized as the symbolization problem or relocatable problem [67, 66], as discussed in Sec. 3.2. `Uroboros` was the first disassembler to be capable of not only disassembling code, but also symbol information from striped binaries automatically [67]. Fig. 4.1 illustrates c2c(code to code), c2d(code to data), d2d(data to data), and d2c(data to code), in total four types of symbol references defined in `Uroboros`. To solve the symbolization problem, `Uroboros` summarized several heuristics to distinguish these four types and thus successfully rewrote all COREUTILS, SPEC INT 2006, and 7 real-world programs. More importantly, after correctly recovering symbol information, `Uroboros` is able to disassemble-reassemble iteration 1,000 times, in which the output of each iteration becomes the input of the next round, with almost zero code size explosion and speed slowdown.

However, manually designed heuristics can hardly achieve binary rewriting across compiler versions because of various optimization and code generation strategies. `Ramblr` proposed more sophisticated heuristics and analyses for symbolization. Based on the `angr` binary analysis platform [61], `Ramblr` is able to rewrite all COREUTILS programs and 143 binaries from the DARPA Cyber Grand Challenge (CGC) on newer compilers (`gcc` 5.4.1 and `Clang` 4.4) with more optimization levels (`O0`, `O1`, `O2`, `O3`, `Ofast`, and `Os`). Although Ramblr went further than the most advanced binary rewriting technique of the time, it did not

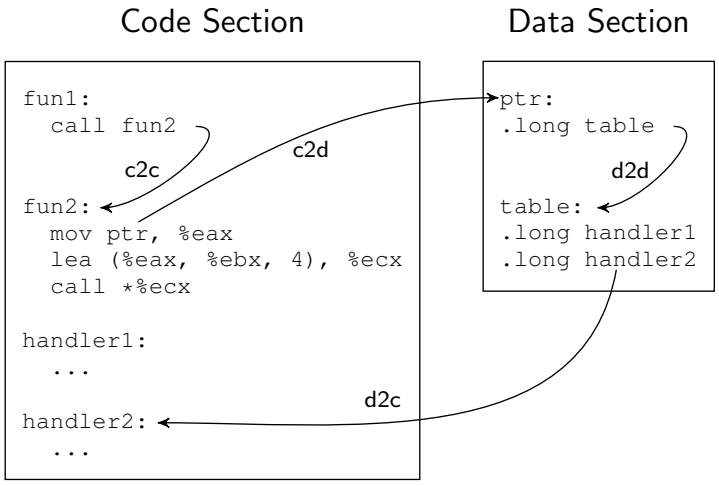


Figure 4.1. Different types of symbol references in assembly code [67].

go beyond the inherent limitations of heuristics and was still error-prone on higher versions of compilers.

4.1.3 PIC Binary Rewriting

Generally speaking, recovering symbol information from stripped binaries has to rely on complicated binary analysis or error-prone heuristics, and thus efficient and error-free static binary rewriting is difficult. With the trend of PIC (Position-Independent Code) becoming the default option of mainstream compilers, some of the subsequent research turns to narrow down their research scopes to 64-bit PIC code only to circumvent the challenging symbolization problem. `RetroWrite` designed a principled symbolization strategy without heuristics [27]. `RetroWrite` leverages relocation information existing in PIC binaries to identify symbolizable constants. More specifically, it recovers symbols by

1) converting targets of control-flow instructions (i.e., calls and jumps) to symbols (recovering code-to-code references),

2) converting the PC-relative addresses to symbols (recovering code-to-code and code-to-data references) to further complete the control flow graph, and

3) converting data relocations to symbols (recovering data-to-data and data-to-code references).

In this way, `RetroWrite` presented a sound and scalable symbolization approach, which makes it robust enough to support multiple security-critical downstream applications, such as `AddressSanitizer` [60] and coverage-guided greybox fuzzing [72].

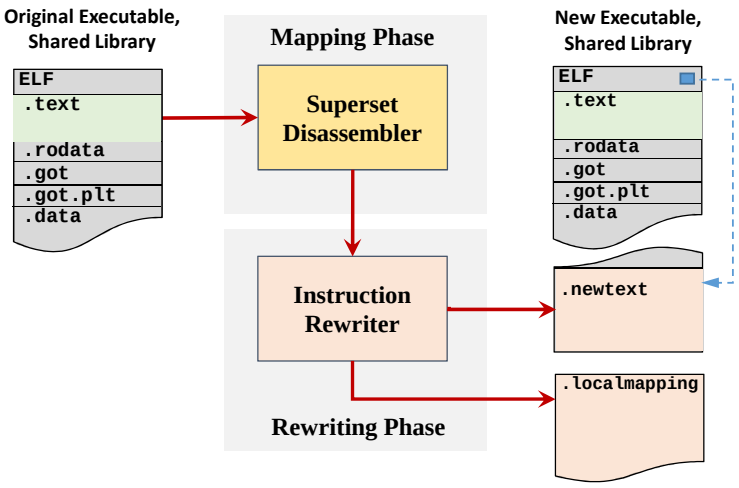


Figure 4.2. Overview of MULTIVERSE [13].

Similarly, `Egalito` uses the relocation information that exists in x86-64 and ARM64 binaries to tackle the symbolization problem. Egalito first lifts disassembled code into the machine-specific, layout-agnostic Egalito IR, then identifies code pointers and reconstructs jump tables on the IR, thus theoretically enabling cross-platform binary rewriting. Their evaluation shows that `Egalito` is adequate to augment hardware/compiler deployment with multiple defense techniques, including a retpoline defense against Spectre [43], a software implementation of Intel's CET [39], and a continuous code randomization defense named JIT-Shuffling [69].

4.1.4 Others

Except for recovering symbol information precisely, another line of research tries to sacrifice code size and performance for stable binary rewriting. For example, MULTIVERSE [13, 52] appends the disassembled code and a mapping from old addresses to new addresses after the original binary as the `.newtext` segment and `.localmapping` segment, as illustrated in Fig. 4.2. The original `.text` segment is kept as read-only data so that recompiled elf binary could read function pointers (code-to-code and data-to-code references) from it and translate the pointers to its new address by looking up the `.localmapping` segment.

On the other hand, E9PATCH introduced a new idea to achieve binary rewriting without control flow recovery [28]. For the purpose of more flexible static x86-64 binary rewriting, E9PATCH came up with a set of instruction-level rewriting methodologies that can insert jumps to trampolines while not changing the layout of the original program. Since this approach does not need a control-flow graph or any binary analysis, it can be easily applied

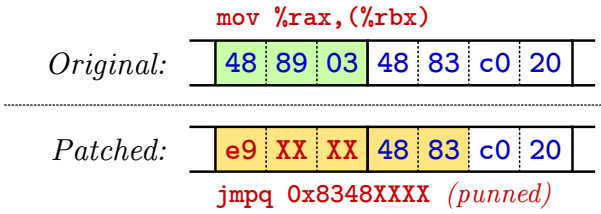


Figure 4.3. Illustration of E9PATCH trampoline [28].

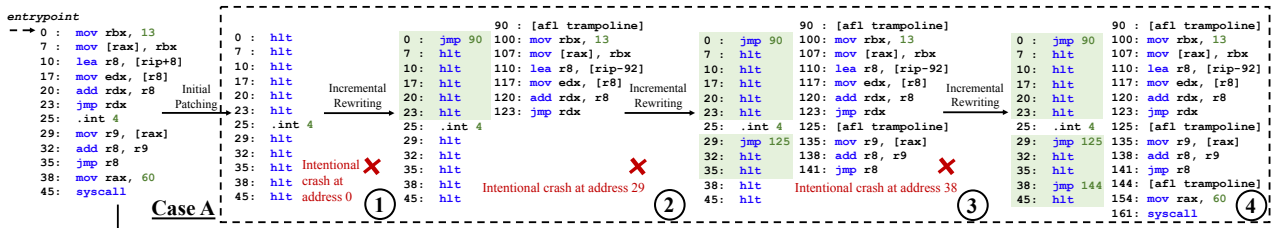


Figure 4.4. One example of STOCHFUZZ rewriting strategies [74].

to large-scale binaries. Fig. 4.3 illustrates one trampoline example of E9PATCH. The original instruction, `mov %rax,(%rbx)`, corresponds to the bytes sequence `48 03 48`, E9PATCH only modify the first 3 bytes to change the instruction to a `jmpq` instruction. The target of this `jmpq` instruction is restricted to a small memory region (`0x8348XXXX`), where the instrumented code could be placed. The E9PATCH approach is robust as no heuristics are used. However, it is still limited as not all x86 instructions could be leveraged with their techniques. According to the evaluation results, this approach could achieve around 99% probability of successful instrumentation.

In addition to all the static methods discussed above, one recent research proposed an interesting idea, STOCHFUZZ, that solves the symbolization problem dynamically [74]. While admitting the difficulties that exist in control flow recovery and symbolization process, this approach does not try to design complicated analyses or heuristics. Instead, it accomplishes symbolization based on observations of runtime behavior, i.e., it dynamically verifies whether an address corresponds to data or code.

To be more specific, this approach proposes an incremental and stochastic rewriting technique that leverages the fuzzing technique to validate assumptions about symbols. It generates many different versions of rewritten binaries then tries to trigger specific behavior with fuzzing runs. Fig. 4.4 illustrate one of the rewriting strategies used in STOCHFUZZ. Given the program listed, the code will be first replace with `hlt` instruction which will case a segment fault, as shown in ①. The segment faults, called *intentional crashes*, indicate that a code block that was not disassembled was found. SOTCHFUZZ starts by fuzzing the rewritten binary. Once the segment fault occurs, the code block that starts from the current address will

be disassembled and placed at a *shadow space* marked as a afl trampoline in the figure. (Note that the instruction at address 110 is `lea r8, [rip - 92]`, in which `rip - 92` corresponds to the original code address `rip + 8`.) After the disassembling, the new binary will be compiled and fuzzed again. Overall, this binary is recompiled with one *incremental* block of disassembled code each time. Therefore this process is called "incremental rewriting". Eventually, all covered blocks will be correctly disassembled by iteratively rewriting and fuzzing without "data or code" confusion.

4.2 Lifting

While disassembly code faithfully expresses the semantics of the program, it operates on the low-level machine architecture, such as registers and memory, and therefore contains much cumbersome information that is unnecessary for understanding the program. Disassembly code is usually refined and lifted into higher-level IR for advanced binary analysis and further understanding of the program. In this section, we introduce and discuss existing IR lifter solutions and related reverse engineering techniques.

4.2.1 LLVM IR Lifting

Static Lifter. SecondWrite is the first reverse engineering tool that lifted disassembly code into a compiler-level IR (LLVM IR) to reuse complex high-level transformations provided by LLVM framework [4, 29]. SecondWrite proposed simple stack frame analysis to deconstruct physical stack into individual abstract stack frames and an algorithm to promote memory locations to symbols aided with Value Set Analysis (VSA) [9]. They also demonstrate the necessity of abstract and symbols for improved dataflow analysis and readability. However, this method does not rely on any debug or symbol information and therefore faces the same challenges mention in Chapter 3. While this approach performs well on their data sets, as compilers have changed and evolved in recent years, it will likely fail on non-trivial binaries.

Dynamic Lifter. A well-developed LLMV IR Lifter would be of great benefit to the whole reverse engineering community. However, although the IR lifter has great application prospects, exploration of the IR lifting technique is still limited due to many difficulties, including variables recovery, types recovery, and translation from assembly code to IR. These difficulties mean that developing a lifter requires a lot of human resources and time. There has been no new lifter technique in academia for a long time after SecondWrite. Until recently, a new dynamic IR lifter named BinRec was proposed [3]. To avoid the trap of variables and types

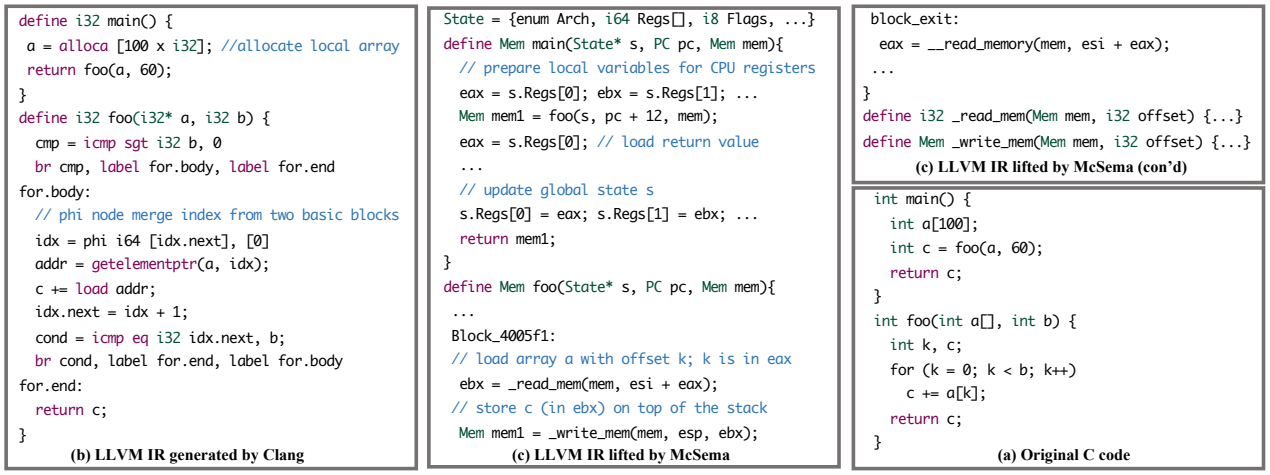


Figure 4.5. A case study comparing LLVM IR lifted by McSema with LLVM IR compiled by Clang.

recovery, BinRec took a different direction by recording the execution trace and later simulating the program execution from the hardware level. Particularly, BinRec is built on top of S^2E [22], a symbolic execution framework running in the QEMU virtual machine [14]. Given a binary program to be lifted, S2E is first used to collect as many distinct execution traces as possible. These traces will be merged and translated into one LLVM IR file, which will stitch with the original program into a new rewritten binary program.

However, this dynamic lifter has two primary defects that significantly reduce the range of its application. First, BinRec employs the *emulation-style* translation strategy that uses LLVM IR to simulate hardware-level program execution. Without recovered higher-level information, such as variables, types, functions, almost all transformations provided by the LLVM framework would not be applied on such IR. Second, the lifted program can only run properly on covered paths, while improving coverage is another frequently studied problem.

Industrial Lifters. Except for academic tools discussed above, some open-source lifters developed by technical companies also achieved great success. For example, McSema, another emulation style lifter developed by Trail of Bits [64], is considered one of the best lifters available today [55]. McSema is also using IR statements to simulate assembly instructions from the hardware level. Specifically, it uses global variables to represent the machine architecture, such as registers and memory, and uses IR statements to read and write these global variables, just like assembly instructions read and write machine architecture.

As illustrated in Fig. 4.5, IR lfited by McSema also does not contain variables, types, function signatures, and other higher-level information, which makes McSema lifted IR not a suitable target for transformation passes provided by the LLVM framework. The difference

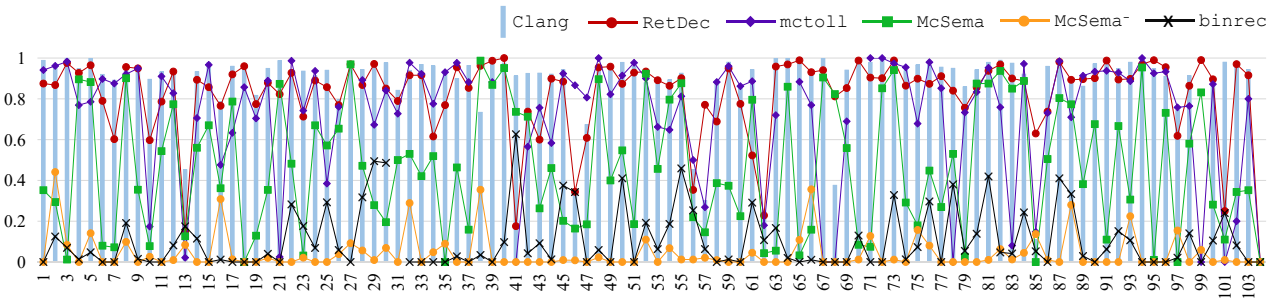


Figure 4.6. Classification analysis results of each tool across all 104 classes of the POJ-104 dataset.

with BinRec is that McSema leverage the cutting-edge commercial reverse engineering tool, IDA [38], for static CFG recovery. Therefore McSema can statically transform and optimize the entire program without worrying about coverage.

On the other hand, Avast [6] developed RetDec [7], as a succinct-style lifter, tries to lift assembly to IR close to compiler-generated IR. However, as we discussed in Sec. 3.3, recovering symbol or debugging information is unsolved. In most cases, RetDec cannot lift non-trivial programs into IR correctly and results in unfunctional rewritten binaries. Therefore, the output of RetDec is only used for tasks that do not require precise semantics, such as machine-learning-based program classification [15] and manual inspection.

4.2.2 Variables and Types Recovery

Although RetDec can hardly produce functional IR for non-trivial programs, it shows great potential in the program classification task. As shown in Fig. 4.6, we apply ncc (Neural Code Comprehension) method [15] on five different lifter settings and compare the classification results with ncc based on compiler-generated IR. Mctoll [50] is another *succinct-style* lifter similar to RetDec, and McSema- represents McSema with all optimizations disabled. Lifters like RetDec and Mctoll show encouraging support for the classification task, whose performance is comparable to compiled IR. Also, we find that lifters like RetDec have only implemented rudimentary analysis passes instead of advanced research products in the variables and types recovery field. So in this section, we will discuss recent variables recovery techniques that may further raise the upper bound of lifters.

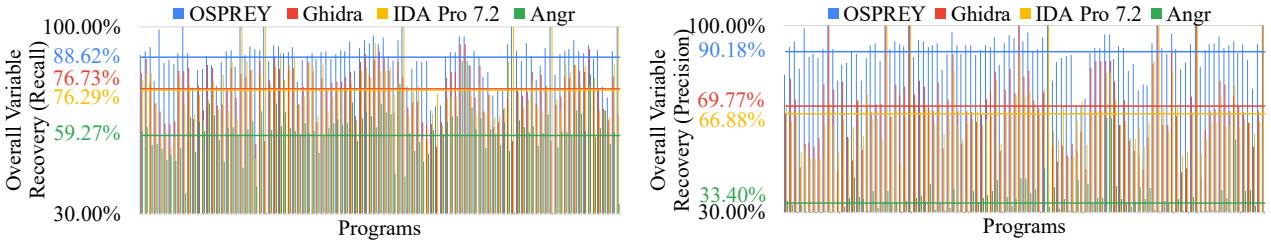
Dynamic Approaches. REWARDS uses a dynamic "type sink" technique to reveal program data structures from binaries automatically [48]. To be more specific, a timestamped attribute will be assigned to each memory location accessed. This attribute is propagated to other memory locations and registers during program execution. Once a type-revealing point or "type sink" that can resolve a variable's type is executed (e.g., the type revealing

instructions, system calls, and library calls), the types of all memory locations sharing the same attribute are resolved. Similarly, another dynamic solution, Howard, aims to recover data structures from memory access patterns in execution traces [62]. These dynamic approaches, while effective, are limited by coverage issues.

Static Approaches. TIE proposes an end-to-end type inference system [47]. It starts by lifting binary to BIL (BAP Intermediate Language) and finding variables with a variation of the VSA method named DVSA. The type system that consists of type inference rules is then used to generate constraints based on the BIL. Finally, the constraints are solved and lead to types as the results. TIE can produce accurate and conservative results, however, the heavy-weight data flow analysis used for variable recovery makes it less practical in large binaries. In the same way, SecondWrite [29] makes a trade-off between accuracy and speed by combining a best-effort VSA variant for points-to analysis with a type-inference engine. Note that accurate types depend on high-quality points-to data, therefore less accurate points-to analysis will lead to a decrease in the accuracy of the results. Also, even original VSA is known to produce a lot of inaccurate results. Comparably, Retypd (**re**gular **ty**pes using **p**ush**d**own) built on top of GrammaTech's machine-code- analysis tool CodeSurfer [8] introduces a principled static type-inference algorithm that supports more advanced types, including recursive types, polymorphism, and subtyping [53].

One of the latest probabilistic variables recovery techniques, OSPREY (recOvery of variable and data Structure by PRobabilistic analysis for strippEd binarY) [73], leverages a more advanced Path Sampling Driven Per-path Abstract Interpretation named BDA [75] instead of VSA for accurate and efficient points-to analysis. Moreover, as inevitable uncertainty in variable and structure recovery may result in contradicting predictions, OSPREY introduces random variables in the type inference system to model the probabilities that a memory location is of a specific type. During type inference, probabilistic constraints are derived from these random variables and will be further solved to produce posterior probabilities as the recovery results. Fig. 4.7 shows that OSPREY achieves around 90% precision, which substantially outperforms state-of-the-art commercial and open-source tools, including IDA [38], Ghidra [54], Angr [61], and Howard [62].

ML-bassed Approaches. Machine learning has also been applied to recover variables and types. DEBIN [37] presents the first machine-learning-based approach to predict debugging information in stripped binaries. It combines an Extremely randomized Tree (ET) [35] classification model with a linear probabilistic graphical model to recover properties of variables (e.g., symbol names, types, and locations). ET is used to extract unknown and known elements (variables) from lifted BAP-IR, then the linear probabilistic graphical model makes



(a) Recall for all variables

(b) Precision for all variables

Figure 4.7. Presision and Recall of OSPREY [73].

joint predictions on the debug information of elements. This approach effectively predicts symbol names and types with 68.8% precision and 68.3% recall on the x64 platform. CATI (Context-Assisted Type Inference) [20] defines a new feature called Variable Usage Context (VUC) for type inference. The basic idea is based on the observation that neighboring instructions are likely to operate the same type of variables. Also, according to the empirical survey, only about 65% of variables have more than 3 related instructions, which means 35% of variables have only 1 or 2 related instructions. Thus to further improve the type inference accuracy, CATI first extracts the VUC feature that contains the target instruction with instruction context, then feeds the VUC feature into Word2Vec [51] model for assembly code embedding, finally, a multi-stage classifier with a convolutional neural network (CNN) is used to predict type for the target variable. The evaluation shows that CATI can achieve 71.2% accuracy for type inference on unseen stripped binaries.

4.2.3 Lifter Verification

Writing a precise binary lifter is extremely difficult even for those heavily tested projects, thus the verification of Lifter is an indispensable part of lifter-related research. MeanDiff [41] is the first lifter verifier that formally checks the semantic equivalence. It tested 3 binary lifters, including BAP [17], PyVEX [1], and BINSEC [12], by translating binary-based IRs (BBIRs) to a Unified Intermediate Representation (UIR) and generating symbolic summaries with data-flow analysis and symbolic execution. Although this method works on some low-level (does not contain variables information) IRs, it cannot be used to test LLVM IR lifters because it is more challenging to define a unified representation for lifted LLVM IR, in the sense that lifters aiming at different applications produce distinct results (i.e., emulation-style and succinct-style).

Dasgupta et al. [25] proposed a new lifter verification framework based on their previous work, in which they provided a complete formal semantics of x86-64 user-level instruction set architecture [26]. Their work is the first single-instruction translation validation frame-

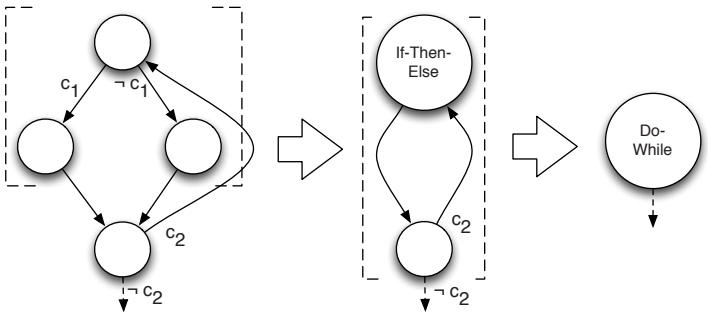


Figure 4.8. Example of structural analysis. [18]

work using the formal semantics of x86-64 and IR. Similar to MeanDiff, it also uses symbolic execution to extract symbolic summaries from x86-64 and IR. With this framework, they proved that McSema is able to correctly translate 2254/2348 functions taken from LLVM's single-source benchmark test suite.

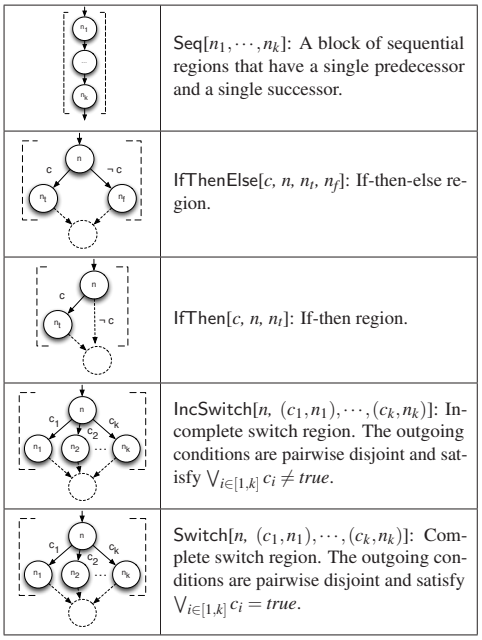
4.3 Decompilation

In general, IR with recovered variables and type information is more analysis-friendly, transformations and analyses provided by LLVM framework could be reused in an automatic way. Nonetheless, sometimes we still need to manually inspect the output of reverse engineering for better understanding (e. g., malware analyses). In such cases, IR involving only simple arithmetic and memory operations is not conducive for human analysts to read and understand. Thus, decompilation is usually used as the last step of reverse engineering to transform IR into high-level programming languages. In this section, we will give a general discussion about decompilation techniques.

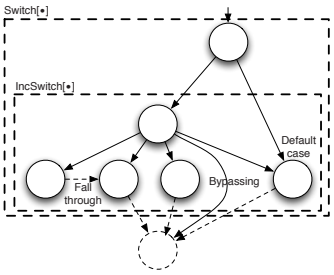
4.3.1 Control Flow Structuring

Based on the lifted high-level IR, the critical step to translate IR to programing language is control flow structuring. As shown in Fig. 4.8, CFG is first reconstructed from IR, then the structural analysis is employed to refine and summarize CFG nodes into high-level control flow structures such as if-then-else constructs and loops.

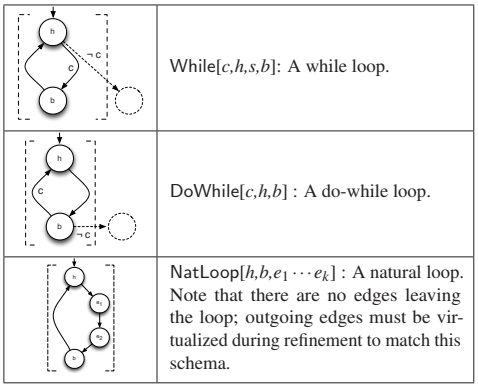
The recovered structure must be completely consistent with the semantics of the original program, thus Phoenix [18] proposed *sematic-preserving* structural analysis that could be safely used in decompilation. Phoenix uses manually crafted graph schemas or patterns to simplify a sub-CFG into a high-level structure, as illustrated in Fig. 4.9. These patterns will



(a)Acyclic regions.



(c)Complete and incomplete switches.



(b)Cyclic regions.

Figure 4.9. Semantics-preserving patterns. [18]

be iteratively applied for structuring. More specifically, Phoenix will iteratively remove an edge from the CFG by emitting a `goto` statement until structuring can progress again.

On top of this idea, DREAM [71] extends the structuring technique with *sematic-preserving transformations* that transform cyclic subgraphs with multiple entries or multiple successors into semantically equivalent single-entry single-successor (acyclic) subgraph. Even more, DREAM presents a *pattern-independent* structuring algorithm that is able to recover all high-level control constructs in acyclic regions. With these two techniques, Dream can produce `goto`-free structured decompiled code. Apart from that, DREAM also produces more compact code compared with Phoenix and the *de facto* industrial standard decompiler Hex-Rays [38].

4.3.2 Meaningful Variable Names Recovery

Besides the structuring techniques that are essential to the correctness of decompiled code, another line of research attempts to recover variable names from decompiled code. For example, DIRE (Decompiled Identifier Renaming Engine) [46] uses an encoder-decoder network to reconstruct semantically meaningful variable names based on the lexical and structural information produced by the decompiler. While this method does not change the

semantics of decompiled code, it can highly increase code readability by achieving 74.3% accuracy in predicting variable names on more than 164k programs collected from Github.

4.3.3 Neural Program Decompiler

With the rise of machine learning, the neural network is also used for decompilation. As an end-to-end neural-based framework for code decompilation, Coda [34] tries to get the decompiled code from assembly code directly without solving the challenges we discussed in Chapter 3. Coda uses an instruction type-aware encoder and a tree decoder to generate an abstract syntax tree (AST) as the code sketch, then updates the code sketch using an iterative error correction machine guided by an ensembled neural error predictor. However, Coda did not compare their results with the best available decompiler, IDA [38]. Instead, they compare with an open-source decompiler named RetDec [7], which cannot represent the state-of-the-art decompilation techniques. More importantly, although Coda claims to be able to preserve both functionality and semantics, the *Token Accuracy* metric they used does not involve semantic or functional checks. Therefore, at this point, we consider that neural network techniques are not ready for the end-to-end decompilation task, given the existing technical conditions. However, we believe that neural networks still have great potential for solving specific problems in software reverse engineering tasks, such as variables recovery and function boundary identification.

CHAPTER 5

ADVANCED TOPICS

As we have discussed the challenges and existing solutions of traditional software reverse engineering, we further discuss several advanced topics related to reverse engineering in this section.

5.1 Smart Contract Decompilation

Smart contracts that run on blockchains are compiled to byte code, finally being executed by Ethereum Virtual Machine (EVM). As lots of valuable tokens and cryptocurrencies are transacted by smart contracts, it is necessary to be able to do reverse engineering on smart contracts for further analyses. Gigahorse [36] presents the state-of-the-art EVM reverse engineering toolchain that can decompile smart contracts from bytecode into a high-level 3-address representation. The source level functions will be reconstructed from the 3-address representation for Solidity-like output. This approach follows a similar pipeline as software reverse engineering on C programs, it first disassembles bytecode into EVM opcodes, then lifts opcode into 3-address IR. Dataflow analyses and heuristics are used during this process to recovery CFG and function signatures. Generally, smart contracts' purpose, functionality, and language features of Solidity are simpler, thus we consider that it is relatively easier to decompile smart contracts compared with decompiling C programs.

5.2 Advanced CFG Recovery

As we discussed in Sec. 3.5, indirect jumps pose a threat to reverse engineering on Object-Oriented Programming (OOP) languages compiled binaries. A precise and scalable binary-level points-to analysis is necessary to solve this problem. While state-of-the-art Value Set Analysis (VSA) is highly conservatives and not scalable, Kim *et al*. proposed BPA (Binary-level Points-to Analysis) [42] to replace VSA for high-precision CFGs construction. By employing memory block generation, BPA achieves higher accuracy and scalability, which points out the possibility of the OOP languages reverse engineering.

5.3 Translation Rules Learning

While almost all lifters implicitly or explicitly define translation rules that translate assembly code to IR, such translation rules are challenging to write in the sense that the developers have to be experts in both assembly language and IR. Although currently, there is no automatic method to generate translation rules without export knowledge, Wang et al. [68, 63] proposed an interesting idea to automatically learn semantic-equivalent translation rules for dynamic binary translation (DBT) system. This approach aims at translating x86 assembly code to arm assembly code, it leverages compiler-generated debug information to find translation rule candidates, then checks the semantic equivalence of candidates through symbolic execution. It is shown that this approach is able to learn tens of thousands of high-quality rules within weeks, which is far more than experts can provide. Although this technique does not involve conversion from assembly to high-level IR, we believe that it points out a possible new direction as automatic learning of translation rules for IR lifting.

CHAPTER 6

OUR WORKS AND POTENTIAL DIRECTIONS

Although existing works have been made significant progress for software reverse engineering, there are still some potential research directions could improve reverse engineering capabilities. In this section, we will discuss some of our current works and possible future research directions.

6.1 Our Works

Decompiler Testing. With over twenty years of development, C decompilers have been widely used in production to support reverse engineering applications. In contrast to this flourishing market, our observation is that in academia, outputs of C decompilers (i.e., recovered C source code) are still not extensively used. We acknowledge that such conservative approaches in academia are a result of widespread and pessimistic views on decompilation correctness. To present an up-to-date understanding regarding modern C decompilers, we test decompilation correctness with the EMI mutation method. Our findings show that state-of-the-art decompilers certainly care about functional correctness, and they are making promising progress. However, some tasks that have been studied for years in academia, such as type inference and optimization, still impede C decompilers from generating quality outputs.

Lifter Benchmarking. Existing research has reported highly promising results that suggest binary lifters can generate LLVM IR code with correct functionality [29]. Beyond that, we conduct an in-depth study of binary lifters from an orthogonal and highly demanding perspective. We demystify the "expressiveness" of binary lifters and reveal how well the lifted LLVM IR code can support critical downstream applications (pointer analysis, discriminability analysis, and decompilation) in security analysis scenarios. Our findings show that modern binary lifters afford IR code that is highly suitable for discriminability analysis and decompilation and suggest that such binary lifters can be applied in common similarity- or code comprehension-based security analysis (e.g., binary diffing). However, the lifted IR code appears unsuited to rigorous static analysis (e.g., pointer analysis).

DNN Decompiler. Due to their widespread use on heterogeneous hardware devices, deep learning (DL) models are compiled into executables by DL compilers to leverage low-level

hardware primitives fully. This approach allows DL computations to be undertaken at a low cost across various computing platforms, including CPUs, GPUs, and various hardware accelerators. We developed a decompiler for deep neural network (DNN) executables named BTD (Binary To DNN). BTD takes DNN executables and outputs full model specifications, including types of DNN operators, network topology, dimensions, and parameters that are (nearly) identical to those of the input models. BTD delivers an automated framework to process DNN executables compiled by different DL compilers and with full optimizations enabled on x86 platforms. It employs learning-based techniques to infer DNN operators, dynamic analysis to reveal network architectures, and symbolic execution to facilitate inferring dimensions and parameters of DNN operators.

6.2 Possible Future Directions

Lifted IR Optimization. We observed that existing lifters follow two distinct designs, the emulation-style lifters produce IR code that is functional, recompilable, but incomprehensible, while the output of succinct-style lifters is relatively analysis friendly but without correctness guarantee. To combine the best of both, we envision that transformation passes provided by the LLVM framework could be applied to optimize emulation-style IR. A large number of semantic-preserving LLVM optimization passes may give us an opportunity to optimize and enrich emulation-style IR expressiveness.

Automatic Translation Rules Learning. As we discussed in Section 5.3, translation rules learning may be a high-potential research direction that would vastly improve existing lifters' performance. However, Translating assembly code to IR is much more complex than dynamic binary instrumentation. Variables and types need to be restored at the same time as translation. With so many related variables and types recovery research, we have to choose one that is compatible with translation rules. Also, as all recovery approaches inevitably have inaccurate results, we need to take fault tolerance into consideration when applying translation rules.

CHAPTER 7

CONCLUSION

This survey presents a systematic overview of the state-of-the-art software reverse engineering techniques. We conclude and discuss the main challenges including, symbolization, variables recovery, types recovery, and control flow recovery. We also present the existing solution for tackling these problems exist in different stages of the reverse engineering pipeline. In the end, we introduce the advanced topics and propose several potential directions to improve the ability of reverse engineering further.

Reverse engineering has been widely used in a variety of security-critical applications and will continue to support security-related tasks. Continued research into more efficient and accurate reverse techniques will benefit the entire security community. We expect this study will help beginning researchers discover more research opportunities and lead reverse engineers to use the latest research output in academia.

BIBLIOGRAPHY

[1] "PyVex," https://github.com/angr/pyvex.

[2] "radare2," http://www.radare.org/r/, 2016.

[3] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida *et al.*, "Binrec: dynamic binary lifting and recompilation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[4] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 295–308.

[5] Angr. (2021) Vex ir. [Online]. Available: https://docs.angr.io/advanced-topics/ir

[6] Avast, "Avast," https://www.avast.com/, 2021.

[7] Avast Software, "RetDec: A retargetable machine-code decompiler," https://retdec.com/.

[8] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86—a platform for analyzing x86 executables," in *International Conference on Compiler Construction*. Springer, 2005, pp. 250–254.

[9] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[10] ——, "Divine: Discovering variables in executables," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2007, pp. 1–28.

[11] ——, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[12] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 165–170.

[13] E. Bauman, Z. Lin, K. W. Hamlen *et al.*, "Superset disassembly: Statically rewriting x86 binaries without heuristics." in *NDSS*, 2018.

[14] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.

[15] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in Neural Information Processing Systems*, vol. 31, pp. 3585–3597, 2018.

[16] Binary Analysis Platform. (2021) Bap ir. [Online]. Available: https://binaryanalysisplatform.github.io/bap/api/odoc/bap/Bap/Std/Bil/index.html

[17] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[18] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 353–368.

[19] J. Caballero and Z. Lin, "Type inference on executables," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–35, 2016.

[20] L. Chen, Z. He, and B. Mao, "Cati: Context-assisted type inference from stripped binaries," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 88–98.

[21] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[22] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[23] C. Cifuentes, *Reverse compilation techniques*. Citeseer, 1994.

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

[25] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, "Scalable validation of binary lifters," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 655–671.

[26] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 1133–1148.

[27] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1497–1511.

[28] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 151–163.

[29] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 51–60.

[30] F. Engel, R. Leupers, G. Ascheid, M. Ferger, and M. Beemster, "Enhanced structural analysis for c code reconstruction from ir code," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, 2011, pp. 21–27.

[31] R. A. Erinfolami and A. Prakash, "Devil is virtual: Reversing virtual inheritance in c++ binaries," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 133–148.

[32] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1075–1092.

[33] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of c++ programs," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 240–243.

[34] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao, "Coda: An end-to-end neural program decompiler," *Advances in Neural Information Processing Systems*, vol. 32, pp. 3708–3719, 2019.

[35] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[36] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1176–1186.

[37] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.

[38] S. Hex-Rays, "Ida pro: a cross-platform multi-processor disassembler and debugger," 2014.

[39] Intel. (2017) Control-flow enforcement technology preview. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[40] O. Katz, R. El-Yaniv, and E. Yahav, "Estimating types in binaries using predictive modeling," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 313–326.

[41] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 353–364.

[42] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," 2021.

[43] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[44] J. Křoustek, P. Matula, and P. Zemek, "Retdec: An open-source machine-code decompiler," 2017.

[45] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.

[46] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[47] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[48] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[49] LLVM. (2021) Llvm ir. [Online]. Available: https://llvm.org/docs/LangRef.html

[50] Microsoft, "llvm-mctoll," https://github.com/Microsoft/llvm-mctoll, 2018.

[51] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[52] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1187–1198.

[53] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 27–41.

[54] N. S. A. (NSA), "Ghidra," https://www.nsa.gov/resources/everyone/ghidra/, 2018.

[55] T. of Bits, "McSema," https://github.com/lifting-bits/mcsema, 2021.

[56] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *arXiv preprint arXiv:2010.00770*, 2020.

[57] T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *International Conference on Compiler Construction*. Springer, 2008, pp. 16–35.

[58] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using logic programming to recover c++ classes and methods from compiled executables," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 426–441.

[59] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 2002, pp. 45–54.

[60] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[61] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[62] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[63] C. Song, W. Wang, P.-C. Yew, A. Zhai, and W. Zhang, "Unleashing the power of learning: An enhanced learning-based approach for dynamic binary translation," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 77–90.

[64] Trail of Bits, "Trail of Bits," https://www.trailofbits.com/, 2021.

[65] M. J. Van Emmerik, *Static single assignment for decompilation*. University of Queensland, 2007.

[66] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again." in *NDSS*, 2017.

[67] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 627–642.

[68] W. Wang, S. McCamant, A. Zhai, and P.-C. Yew, "Enhancing cross-isa dbt through automatically learned translation rules," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 84–97, 2018.

[69] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 367–382.

[70] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 133–147.

[71] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations." in *NDSS*. Citeseer, 2015.

[72] M. Zalewski, "American fuzzy lop," 2014.

[73] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 813–832.

[74] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 659–676.

[75] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.