Reassembleable Disassembling

Shuai Wang, Pei Wang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
{szw175, pxw172, dwu}@ist.psu.edu

Abstract

Reverse engineering has many important applications in computer security, one of which is retrofitting software for safety and security hardening when source code is not available. By surveying available commercial and academic reverse engineering tools, we surprisingly found that no existing tool is able to disassemble executable binaries into assembly code that can be *correctly assembled back* in a fully automated manner, even for simple programs. Actually in many cases, the resulted disassembled code is far from a state that an assembler accepts, which is hard to fix even by manual effort. This has become a severe obstacle. People have tried to overcome it by patching or duplicating new code sections for retrofitting of executables, which is not only inefficient but also cumbersome and restrictive on what retrofitting techniques can be applied to.

In this paper, we present UROBOROS, a tool that can disassemble executables to the extent that the generated code can be assembled back to working binaries *without manual effort*. By empirically studying 244 binaries, we summarize a set of rules that can make the disassembled code *relocatable*, which is the key to *reassembleable disassembling*. With UROBOROS, the disassembly-reassembly process can be repeated thousands of times. We have implemented a prototype of UROBOROS and tested over the whole set of GNU Coreutils, SPEC2006, and a set of other real-world application and server programs. The experiment results show that our tool is effective with a very modest cost.

1 Introduction

In computer security, many techniques and applications depend on binary reverse engineering, i.e., analyzing and retrofitting software binaries with the source code unavailable. For example, software fault isolation (SFI) [33, 46, 2, 19, 18] rewrites untrusted programs at the instruction level to enforce certain security policies. To ensure program control-flow integrity (CFI, meaning that program execution is dictated to a predetermined control-flow graph) [1, 4, 43, 17, 29, 37] without source code, the original control-flow graph must be recovered from a binary executable and the binary must be retrofitted with the CFI enforcement facility embedded [50, 49]. Symbolic taint analysis [34] on binaries must recover assembly code and data faithfully. The defending techniques against return-oriented programming (ROP) attacks also rely on binary analysis and reconstruction to identify and eliminate ROP gadgets [44, 9, 47, 22, 39].

Despite the fact that many security hardening techniques are highly dependent on reverse engineering, flexible and easy-to-use binary manipulation itself remains an unsolved problem. Current binary decompilation, analysis, and reconstruction techniques still cannot fully fulfill many of the requirements from downstream. To the best of our knowledge, there is no reverse engineering tool that can disassemble an executable into assembly code which can be *reassembled* back in a fully automated manner, especially when the processed objects are commercial-off-the-shelf (COTS) binaries with most symbol and relocation information stripped.

We have investigated many existing tools from both the industry and academia, including IDA Pro [24], Phoenix [42], Dagger [12], MC-Semantics [32], SecondWrite [3], BitBlaze [45], and BAP [8]. Unfortunately, these tools focus more on recovering as much information, such as data and control structures, as possible for analysis purpose mainly, but less on producing assembly code that can be readily assembled back without manual effort. Hence, none of them provide the desired disassembly and reassembly functionality that we consider, even if the processed binary is small and simple.

Due to lack of support from reverse engineering tools, people build high-level security hardening applications based on partial binary retrofitting techniques, including binary rewriting tools such as Alto [35], Vulcan [16],

Diablo [13], and binary reuse tools such as BCR [11] and TOP [48]. We consider binary rewriting as a partial retrofitting technique because it can only instrument or patch binaries, thus not suitable for program-wide transformations and reconstructions. As for binary reuse tools, they work by dynamically recording execution traces and combining the traces back to an executable, meaning the new binary is only an incomplete part of the original binary due to the incomplete coverage of dynamic program analysis.

Partial retrofitting has notable drawbacks and limitations:

- Patch-based rewriting could introduce non-negligible runtime overhead. Since the patch usually lives in an area different from the original code of the binary, interactions between the patch and the original code usually require a large amount of control-flow transfers.
- Patch-based rewriting usually relocates instructions at the patch point to somewhere else to make space for the inserted code. As a result, it requires the affected instructions to be relocatable by default.
- Instrumentation-based rewriting expands binary sizes significantly, sometimes generating nearly double-sized products.
- Binary reuse often requires a binary component to be small enough for dynamic analysis to cover; otherwise the correctness cannot be guaranteed.

Having investigated previous research on binary manipulation and reconstruction, we believe that it could be a remarkable improvement if we are able to automatically recover the assembly from binaries and make the assembly code ready for reassembly. When a binary can be reconstructed from assembly code, many high-level and program-wide transformations become feasible, leading to new opportunities for research based on binary retrofitting such as CFI, diversification, and ROP defense.

Our goal is quite different from previous reverse engineering research. Instead of trying to recover high-level data and control structures from program binaries which helps binary code analysis, we aim at a more basic objective, i.e., producing assembly code that can be readily reassembled back without manual effort, which we call the *reassembility* of disassembling. Although the research community has made notable progress on binary reverse engineering, reassembility is still somewhat a blank due to lack of attention. In this sense, our contribution is complementary to existing work.

With that said, we believe that the technical challenge is also a cause for the deficiency in binary reassembly support from existing tools. We have confirmed that the key to reassembility is making the assembly code *relocatable*. Relocation is a linker concept, which is basically for ensuring program elements defined in different source files can correctly refer to each other after linked together. Being relocatable is also a premise for supporting program-wide assembly transformations. In COTS binaries, however, the information necessary for making disassembly results relocatable is mostly unavailable. There has been research trying to address the relocation issue [11, 48, 26], but existing work mostly relies on dynamic analysis which is unlikely to cover the whole program.

In this paper, we present UROBOROS, a disassembler that does reassembleable disassembling. In UROBOROS, we develop a set of methods to precisely recover each part of a binary executable. In particular, we are the first to be capable of not only recovering code, but also data and meta-information from COTS binaries without manual effort. We have implemented a prototype of UROBOROS and tested it on 244 binaries, including the whole set of GNU Coreutils and the C programs in SPEC2006 (including both 32-bit and 64-bit versions). In our experiments, most programs reassembled from UROBOROS's output can pass functionality tests with negligible execution overhead, even after repeated disassembly and reassembly. Our preliminary study shows that UROBOROS can provide support for program-wide transformations on COTS binaries.

In summary, we make the following contributions:

- We initiate a new focus on reverse engineering. Complementary to historical work which mostly focuses on recovering high-level semantic information from binary executables or providing support for binary analysis, our work seeks to deliver *reassembility*, meaning we disassembles binaries in a way that the disassembly results could be directly assembled back into working executables, without manual edits.
- We identify the key challenge is to make the disassembled program *relocatable*, and propose our key technique to recover references among immediate values in the disassembled code, namely "symbolization".
- With reassembility, our research enables direct binary-based transformation without resort to the previously used *patching* method, and can potentially become the foundation of binary-based software retrofitting.
- We implement a prototype of UROBOROS and evaluate its strength on binary reassembly. We applied our technique to 244 binaries, including the whole

set of GNU Coreutils and SPEC2006 C binaries. The experiment results show that our tool does correct disassembly and introduces only modest cost.

- Our disassembler produces "normal" assembly code in the sense that binaries reassembled from UROBOROS's output assembly can again be disassembled (and hence the name UROBOROS[1]), or be used to accomplish other reverse engineering tasks. We verify this by repeating the disassemble-reassemble loop for thousands of times on different binaries.

The remainder of the paper is organized as follows. We first discuss the related work and challenges in §2 and §3, respectively. We then present the design and implementation of UROBOROS in §5. The experimental results are presented in §6, followed by some discussions in §7. We conclude the paper in §8.

2 Related Work

This section reviews literature on binary disassembly, binary rewriting, and binary reuse.

2.1 Disassembly

As aforementioned, there is no disassembler known to us that can generate working assembly code from binaries whose symbol and relocation information is stripped. IDA Pro [24] is considered as the best commercial disassembler available on the market. It can decode binaries into assembly and further decompile assembly into C code for program analysis. However, the assembly code produced by IDA Pro cannot be directly used as the input of any assembler. As stated in its manual [21], assembly code produced by IDA Pro is meant for analysis and cannot be directly reassembled or recompiled.

SecondWrite [3] leverages multiple static analysis techniques to lift binaries into LLVM IR. It is reported that the recovered LLVM IR can be converted back into C code given the LLVM's IR-to-C backend. However, it is unclear to us how SecondWrite symbolizes the data sections and recovers the meta-data information of the binaries. The paper does not contain an evaluation on this recompilation functionality. Moreover, the IR-to-C backend has been removed from LLVM release since 3.1, because it is not mature enough to handle non-trivial programs [30].

Dagger [12] is another tool that translates native code into LLVM IR, but the implementation is far from complete. There is a pre-release version available online. We tried to use it to decompile a simple binary (compiled from a C program with only empty main function). The decompiler reported several errors and generated an LLVM IR file which cannot be compiled back into binary due to lack of some symbol definitions.

MC-Semantics [32] is yet another tool for native code to LLVM IR translation. We used MC-Semantics to decompile some quickly written mini programs. Although the code produced by MC-Semantics can be made binaries, the execution results of these binaries are not the same as the originals, which we believe is due to incorrect symbol references. In addition, different from previously reviewed work, MC-Semantics works at the scale of object files rather than executables. Lacking the ability to handle linked binary programs narrows its scope of application.

BAP [8] is a binary analysis platform that comes with a disassembler. It can lift assembly code to a BAP-defined high-level intermediate representation that can be further analyzed statically. Several reverse engineering tools have been built based on BAP, including the C type recovery tool TIE [28] and the C control-flow recovery tool Phoenix [42]. Although BAP provides solid support for binary analysis, the strength of its disassembler is also limited to analysis only.

There could be multiple reasons that existing tools fail on reassembling. One reason is the technical challenges such as separating code and data, symbolizing the data sections, etc. The other reason could be the difference in the design goals. Most existing tools aim to produce more readable code or code that can be analyzed, not for the purpose of translation and reassembly. We emphasize that the ability to reassemble the output from a disassembler can provide an enabling infrastructure, facilitating further research.

2.2 Binary Rewriting

Binary rewriting techniques can be either static or dynamic. Static binary rewriting is widely used in security hardening such as control-flow hijacking mitigation [47], software control-flow integrity enforcement [50, 49], and binary instrumentation [3, 35, 13, 16]. Most static binary rewriting tools make strong assumptions on the input binaries. For example, Vulcan [16], Alto [35, 13], and Diablo [13] require binaries to be compiled from specific compilers or require symbol information not stripped. SecondWrite [3] can patch binaries with new code and data, but the original content in the binary shall remain unmodified.

As aforementioned, typical static binary rewriting has to relocate instructions at the patch point to make room for newly inserted code. In order to make sure that the rearranged instructions can be relocated while still pre-

[1] Uroboros is a symbol depicting a serpent eating its own tail.

serving program semantics, a stub-based idea is adopted to redirect control flow from the original location to the relocated new place at run time [14, 50, 47]. Control transfer instructions, i.e., stubs, are inserted at memory addresses that are pointed to by some code pointers. This strategy broadens the application scope of binary rewriting tools. However, there could be a large amount of stubs inserted, thus incurring notable execution overhead and size expansion on the rewritten binaries.

Dynamic binary rewriting tools, such as Pin [31] and DynamoRIO [7], can trace the execution of a binary and instrument or patch the program on the fly. Dynamic rewriters are able to handle COTS binaries, whereas with the cost of considerable performance penalty. Also, dynamic binary rewriting requires the rewriter itself to be shipped with or embedded into the target binaries.

Dyninst [10, 20] is a tool that features both static and dynamic binary rewriting. It supports performance measurement and computational steering. It can disassemble the stripped binaries and instrument them statically or dynamically, but does not deliver reassembleable disassembling either.

2.3 Binary Reuse

Binary reuse is mostly based on dynamic analysis. One of the representative binary reuse tools is BCR [11]. BCR extracts and reuses functions from binaries with a hybrid approach. BCR first executes binaries in a monitored environment and records execution traces and memory dumps. Binaries are then statically disassembled starting from the entry point. In the disassembly process, the dynamically collected information is used to resolve the destinations of indirect branches. In the end BCR manages to extract a "closure" of code reachable from the entry point which can be reused by other programs. Clearly, the correctness of the reused code cannot be guaranteed if BCR does not cover all feasible execution paths.

In addition to BCR, there are other binary reuse tools that employs similar basic ideas, such as Inspector Gadget [26] and TOP [48]. While these tools have made improvements in different aspects, the fact that they all rely on dynamic analysis leads to the incompleteness issue, more or less. In general, these tools can only do partial binary retrofitting.

3 Challenges

We have briefly discussed the technical challenges for developing a disassembler which can deliver *reassembility*. In this section, we discuss these difficulties in more details. In this research, we assume that the binaries to disassemble are stripped COTS binaries, namely binaries without any relocation information or symbols, except those necessary for dynamic linking. We also assume that the binaries are compiled from unobfuscated C programs, without self-modifying features. The target hardware architectures of the binaries are x86 and x64. The binary executable format is the Executable and Linkable Format (ELF).

3.1 Raw Disassembly

In this paper, raw disassembly is referred to as the process of parsing the binary form of a program to its raw textual representation. The difficulty of raw disassembly can vary a lot in different situations. In the most general case, this problem is undecidable. One of the reasons is that the problem of statically determining the addresses of indirect jumps is undecidable [23]. Furthermore, the existence of advanced program features such as self-modifying code makes the problem harder. Another issue is that current computer architectures do not distinguish code and data, and there is no easy way for a raw disassembler to distinguish them either. This problem is further worsened by the variable-length instruction encoding used by, for example, the x86 instruction set architecture.

However, with years of intensive effort on improving related techniques, the state of the art can already reach a very high success rate when disassembling binaries compiled from practical legitimate C source code by mainstream compilers. A recent paper by Zhang et al. [50] proposed a novel raw disassembly method which combines two existing disassembly algorithms together. We reimplemented this algorithm and applied it to our evaluation set which includes 244 binaries. No errors were reported by the raw disassembler and subsequent evaluation also verified the correctness of this algorithm on our evaluation set. As a result, we do not consider raw disassembly, or binary decoding, as a major challenge to address in this research.

3.2 Reassembly

Successfully decoding the binaries is only the first step to the goal of this research. Ideally, binary reverse engineering tools should be able to support at least the following process:

- The reverse engineering tool disassembles the original binary into assembly code.
- Users can perform static analysis on the disassembled program.
- Users can perform transformations on the disassembled program.

- The transformed program can be assembled back into an usable binary executable, with all transformation effects retained.

Although it may not be obvious, the feasibility of the first three steps does not naturally imply the feasibility of the last step. There have been reverse engineering tools or platforms that can (partially) enable the first three steps [8, 45], but support for reassembly is still a blank.

As mentioned in the introduction, making the assembly code relocatable is the crux of reassembility. Figure 1 is an artificial example comparing relocatable and unrelocatable assembly code. In COTS binaries, information required for making disassembly results relocatable is unavailable. Most program transformations inevitably change binary layouts, but a reverse engineering tool has only very limited control over how the linkers assign memory addresses of the program elements, leading to situations illustrated by Figure 1. Note the memory cell located at address `0xc0` in the original memory, which is possibly a global variable. The raw disassembly process does not recognize the concrete value `0xc0` in the code as a reference. Thus when this unrelocatable assembly code is reassembled, the resulting binary will very likely be defected because the content of the memory cell at `0xc0` in the original binary may not be placed the same address in the new binary. In the relocatable assembly, however, the data originally living at `0xc0` is given a symbolic name, and the concrete address `0xc0` is replaced by a reference to this name. This is why relocatable assembly can be reassembled into a working executable.

As suggested by the example, if a reverse engineering tool seeks to reassemble the transformed assembly code into a working executable, it has to identify program elements whose addresses could possibly change in the new binary, and lift concrete memory addresses referring to them to abstract symbolic references. Obtaining relocatable assembly from a COTS binary is non-trivial because very little auxiliary information in the binary can be utilized to help identify references among concrete values. Essentially, the problem can be generalized as the following: given an immediate value in the assembly code (either in a code section or data section), is it an memory address or a constant? Although this looks like a typical type analysis problem, in the context of binary reassembly it becomes much more challenging. From a static point of view, since most machine assembly languages are untyped, type inference is difficult in the first place. Compared to high-level programming languages, assembly languages lack explicit syntax for denoting procedure boundaries and basic control-flow logic, making static analysis even more difficult. What is worse, many references live in the data sections, some of which are indirectly referred to by the code via numerous reference hops. At present, most proposed program analysis techniques, either static or dynamic, are code oriented, lacking the capability of analyzing the property of a given data chunk. Finally, reassembly has almost zero tolerance for type inference errors, because a single false positive or false negative can place the reassembled binary in a non-functional state.

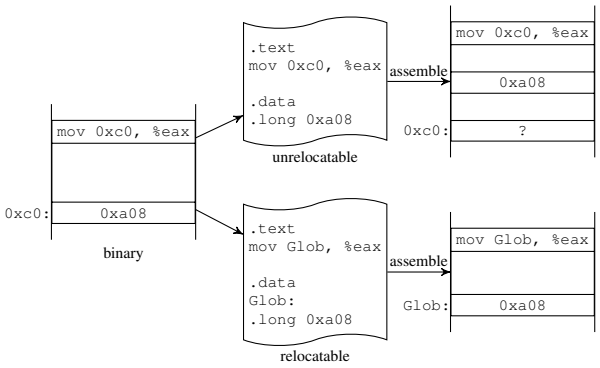

Figure 1: Relocatable and unrelocatable assembly code

Solving the relocatable problem in binary disassembly is the main purpose and contribution of this paper. In the rest of the paper, we call the process of identifying references among immediate values in the raw assembly the process of "symbolization". To distinguish the concept from the traditional meaning of disassembling, we call our work *reassembleable disassembling* that generates relocatable assembly code.

In addition to relocation information, a full-fledged disassembler also needs to recover some meta information to make the reassembly feasible. Meta-data sections in a binary executable provide information to direct some link-time and runtime behavior of the program. They should also be recovered properly in order to ensure the reassembled binaries are semantic-equivalent to the originals.

4 Symbolization

This section describes the symbolization problem in detail and presents our solution.

4.1 Classification

There are four types of symbol references that we need to identify for reassembility. The classification is based on two criteria—where a reference lives and where a reference points. Basically, we divide the binary into two parts, i.e., the code sections and the data sections, whose contents are as suggested by their names. For ELF binaries on Unix-like platforms, typical code sections include

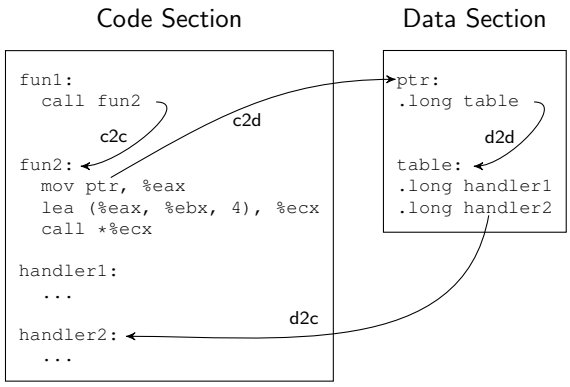

Figure 2: Different types of symbol references in assembly code

`.text` and `.init` etc. Typical data sections include `.data`, `.rodata`, `.bss`, etc. A symbol reference can live in either code sections or data sections, and can point to either code sections or data sections as well, leading to a total of four types. Figure 2 is an example showing all four types of symbol references. We give each of them a short name, i.e., c2c, c2d, d2c, and d2d references.

4.2 Method

When it comes to solving the symbolization problem, we have considered various potential solutions. Due to the reasons listed in §3.2, we conclude that no existing program analysis technique can handle the symbolization problem in our special context. Hence, we decide to turn to another direction. In this work, we identify the immediate values which are actually symbol references by applying several matching rules inferred from our study on a large amount of binaries. Although some of these strategies may not seem exciting at the first sight, they work surprisingly well in our evaluation on 244 binaries compiled from C code.

Since we are solving the symbolization problem in an empirical way, the matching strategies are all based on certain assumptions. Depending on whether an assumption is accepted or not, different rules are applied for symbolization. We now introduce the assumptions and the corresponding symbolization strategies.

At the point of symbolization, we assume that we have already obtained the raw assembly decoded from binaries using the algorithm by Zhang et al. [50], so we can get all immediate values that appear in a binary. There are two kinds of immediate values—constants used as instruction operands and the byte stream living in data sections. Among all these immediate values, some can be excluded from being considered for symbolization at the first place. Unless a program intentionally causes memory access errors, which is rarely the case, an immediate value can be a reference to symbols *only if* this value falls in the address space allocated for the binary. For a binary of reasonable size, the utilization of address space is usually sparse, so there is a wide range of address space which is actually invalid.

Assuming all immediate values are potential symbol references, we can filter out obviously invalid references based on their target addresses. According to our symbol reference classification in §4.1, a reference can only point to code sections or data sections; especially, if a reference points to code sections, the destination must be the starting address of some instruction. Our study on 244 binaries shows that this simple filter is sufficient to identify c2c and c2d symbol references with full correctness.

The really challenging part is data section symbolization, i.e., identifying d2c and d2d references. The first step of data section symbolization is to slice the data sections, which are continuous areas of binary bytes, into individual values of different lengths. Since the raw disassembly process does not assign the data sections any semantics, there is no ready-made guidance on how they should be sliced. Regarding this problem, we introduce the first assumption which is about binary layout:

> (A1) *All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.*

Since unaligned memory accesses cause considerable performance penalty, compilers tend to keep data aligned by its size. For data alignment, compilers can even sacrifice memory efficiency by inserting padding into data sections. With that said, A1 stays as an assumption because occasionally programmers do want non-aligned data layout. For example, the "packed" attribute supported by GCC allows programmers to override the default alignment settings.

If we accept assumption A1, only n-byte long values which are also n-byte aligned in data sections are considered for symbolization. Alternatively with A1 rejected, all n-byte long memory content in data sections are considered for symbolization. This is implemented as an n-byte sliding window which starts from the beginning of a data section and scans through the entire section in a *first-fit* manner. Each time the sliding window moves forward 1 byte and check the value of the covered bytes. If the value fulfills the basic requirements for being a d2d or d2c reference, it will be considered for symbolization and the sliding window advances n bytes forward. In case that the value does not meet the requirements, the sliding window moves forward 1 byte only.

In addition to assuming the characteristics of binaries, making assumptions on user requirements for our tool also helps improve its performance. As stated earlier,

the goal of symbolization is to make assembly code relocatable so that users can perform program-wide transformations on the assembly and then assemble it back to a working executable. From our experience, most transformations on assembly only touch the instructions without modifying the original data. If we make the following assumption

> (A2) *Users do not need to perform transformation on the original binary data.*

then we can keep the starting addresses of data sections the same as their old addresses when performing reassembly, by providing a directive script to the linker. In this way, we can ignore d2d references during symbolization simply because we do not need them to be relocatable anymore. Thus, with A2 accepted, only the immediate values that fall within code sections (d2c references) are considered for symbolization. Contrarily, without deterministically fixing the starting addresses of data sections in the new binary, the immediate values that fall within either code sections or data sections are considered for symbolization.

We want to avoid symbolizing d2d references because they are used in a very flexible manner. On the other hand, there are more common patterns in d2c references which can be exploited by our symbolization method. We summarize the patterns with the following assumption:

> (A3) *d2c symbol references are only used as function pointers or jump table entries.*

By accepting A3, an n-byte value in data sections is lifted to a d2c reference if it is the starting address of some function, or it forms a jump table together with other n-byte values adjacent to it. Otherwise with A3 rejected, an n-byte data section value is symbolized whenever it is within the address space of code sections.

When A3 is taken, we will need to know whether a code section address is the start of a function. We also need to clarify what a jump table would be in the binary form. Identifying function beginnings in a binary is not a new research topic. Based on machine learning techniques, recent research [6] can reportedly identify function starting addresses with over 98% precision and recall. To avoid reinventing the wheel, we assume we have already known all the function start addresses. Since the binaries used in our research are all compiled from source code, we are able to get the ground truth by controlling the compilation and linking process.

Regarding the identification of jump tables, our algorithm is as follows:

- *Jump table start.* We traverse the data sections from the beginning to the end. If the address of an n-byte value is referred to by an instruction as the operand, it is considered as the first entry of a new jump table.
- *Jump table entry.* If an n-byte value follows an already identified jump table entry, this value is also considered as an entry as long as it refers to instructions within the same function that previous entries point to.

The three assumptions A1, A2, and A3 are the basics of our symbolization method. With different choices of an assumption being applied or not, we can derive different strategies when processing a binary. §6 has a detailed evaluation on the correctness of reasonable combinations of these assumptions.

5 Design and Implementation

5.1 Overview

The architecture of UROBOROS is shown in Figure 3. UROBOROS consists of two main modules—the disassembly module and the analysis module. The disassembly module decodes instructions with raw disassembling (§5.2) and dumps the data sections. The analysis module symbolizes memory references in both code and data sections (§4) and recovers the meta-information from the dumped content (§5.4). UROBOROS also recovers part of the control-flow structures from direct transfers so that it provides basic support for program-wide transformation (§5.3).

The disassembly module employs an interactive process to validate disassembled code from a linear disassembler. The linear disassembler decodes the code sections and dumps out all data and meta information sections. A validator is then invoked to correct disassembly errors due to "data gaps" embedded inside code sections. The details are presented in §5.2.

After the raw disassembly is over, the dumped code, data, and meta-data are sent to the analysis module. This module identifies symbol references among immediate values in the code and data. As elaborated in §4, we propose three assumptions for reassembleable disassembling. The corresponding strategies are implemented in UROBOROS to guide the symbolization process. UROBOROS can be configured to utilize different combinations of assumptions for symbolization. We give a detailed evaluation on the correctness of different strategies in §6.1.

Given the symbolized instructions, the analysis module also partially recovers the control flows based on direct control-flow transfers. With the relocatable assembly and the basic control-flow structures, users of UROBOROS can easily perform advanced program anal-

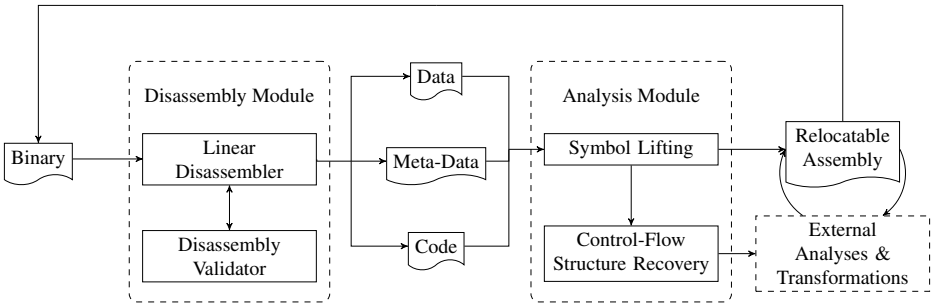

Figure 3: The architecture of UROBOROS

ysis and program-wide transformations before they assemble the code back to binaries.

Finally, we emphasize that the assembly code generated and transformed by UROBOROS can be directly assembled back as a working binary by normal assemblers. In particular, the binary output is indeed a *normal* executable file without any abnormal characteristics such as patched or duplicated sections. Therefore, the reassembled binary can be disassembled again by UROBOROS or be processed by other reverse engineering tools.

We have implemented a prototype of UROBOROS in OCaml and Python, with a total of 13,209 lines of code. Our prototype works for both x86 and x64 ELF binaries.

5.2 Disassembly

In our prototypical implementation, the linear disassembler employed by UROBOROS's disassembly module is `objdump` from GNU Binutils. We implement an interactive disassembly process originally proposed in BinCFI [50].[2] In this process, the disassembler communicates with a validator which corrects disassembly errors due to "data gaps" between adjacent code blocks. The interactive procedure is as follows:

- `objdump` tries to decode the input binary for the first time.
- The validator examines the output and check if there are explicit errors reported by `objdump`. In case there are no errors, the raw disassembly process terminates. Otherwise, the validator assumes the errors are caused by data embedded in code and computes the upper and lower bounds of identified "data gaps".
- With the computed range of identified "gaps", the validator guides `objdump` to decode the binary again, with those "gaps" skipped.
- Repeat this decode-validate process until no error occurs or the running time of the whole process reaches a time limit specified by users.

We leverage three rules proposed in BinCFI to validate the disassembly results and locate the data "gaps", i.e., "invalid opcode", "direct control transfers outside the current module", and "direct control transfer to the middle of an instruction". Since identifying bounds of each data gap can rely on the control-flow information of decoded instructions, the validator occasionally leverages UROBOROS's analysis module to retrieve the control-flow information.

5.3 Support for Program Transformation

UROBOROS provides basic support for program-wide transformations by partially recovering control-flow structures of the decoded instructions. We collect all the control transfer instructions to divide each function into multiple basic blocks. Control-flow graphs are rebuilt on top of these basic blocks. As a prototype, UROBOROS currently only processes direct control transfers. Regarding the intractable indirect transfers, a potential solution is to use value set analysis (VSA) [5] for destination computation. We leave including indirect control transfers in the CFG as future work.

5.4 Meta-Information Recovery

UROBOROS recovers the program-linkage table (PLT) and the export table in ELF binaries. The PLT table supports dynamic linkage by redirecting intra-module transfers on its stubs to external functions. As the base address of the PLT table can change after reassembling, we translate the memory references to PLT stubs to their corresponding external function names, and let the linker to rebuild the PLT table with correct memory references during link time. In particular, this table is dumped out from the input binary and parsed into multiple entries, each containing the memory address of a PLT stub with

[2]The BinCFI tool is available open source. We choose to reimplement the algorithm to make the codebase of UROBOROS more consistent such that it is fully automated and easy to extend. We refer readers to BinCFI [50] for the details of the disassembly process.

its corresponding function name. Next, we scan the program and identify the addresses that match to a table entry. These addresses are then replaced by the corresponding function name.

Symbols need to be "exported" so that other compilation units can refer to them. The exported symbols together with their memory addresses are recorded in the export table. As ELF binaries do not keep a standalone export table, we construct this table by searching for all global objects in the symbol table. The symbol name of each entry and its memory address are then kept in a map. The export table can help identify functions and variables that are only referred to by other compilation units. We iterate each entry of the export table to insert symbols and `.globl` macros to the corresponding addresses.

For typical ELF binaries compiled from C code, `.eh_frame` and `.eh_frame_hdr` sections are used by compilers to store information for some rarely-used compiler-specific features, such as the "cleanup" attribute supported by GCC. For these sections, we dump the content out and directly write them back to the output. These sections are also used to store exception information for C++ programs. Regarding this, we have a related discussion in §7.

5.5 Position Independent Code

Position independent code (PIC) typically employs a particular routine to obtain its memory address at run time. This address is then added by a fixed memory offset to access static data and code. According to our observation, the routine below is utilized by PIC code in 32-bit binaries to achieve relative addressing.

```
804C452: mov (%esp),%ebx
804C456: ret
```

PIC code invokes this routine by a `call` instruction, and register `ebx` is then assigned the value on top of the stack, which equals the return address. UROBOROS identifies this instruction pattern, traces the usage of `ebx`, and rewrites the instructions that add `ebx` with memory offsets to a relocatable format.

An example is shown in Figure 4. Once we identify a `call` instruction targeting the above sequence, we calculate the absolute address by adding `0x804c466` with offset `0x2b8e`, which equals `0x804eff4`. By querying the section information from ELF headers, `0x804eff4` equals the starting address of `.got.plt` table, and we rewrite offset `0x2b8e` to the corresponding symbol, which is `_GLOBAL_OFFSET_TABLE` in this case.

Theoretically PIC could use other patterns besides the above sequence to obtain its own memory address; the above instruction sequence is, however, the only PIC pattern we encountered after testing a broad range of real world applications (compiler and platform information is disclosed in § 6).

```
804c460: push %ebx
804c461: call 804c452
804c466: add $0x2b8e,%ebx
804c46c: sub $0x18,%esp

804c460: push %ebx
804c461: call S_0x804C452
804c466: add $_GLOBAL_OFFSET_TABLE_,%ebx
804c46c: sub $0x18,%esp
```

Figure 4: PIC code reuse

As for x64 architectures, RIP-relative [25] memory references allow assembly code to access data and code relative to the current instruction by leveraging the `rip` register and memory offsets, which makes the implementation of PIC more flexible. In the raw disassembly output, instructions utilizing this mode are commented by `objdump` with the absolute addresses they refer to. We identify the comments, symbolize the memory offsets, and insert labels to the corresponding absolute addresses.

5.6 Redundancy Trim

When a binary is dynamically linked to libc, the prologue and epilogue functions of the library are automatically added to the final product. UROBOROS attempts to support multiple iterations of the disassemble-reassemble process. Each time the binary is assembled, a new copy of the prologue and epilogue functions are inserted, which unnecessarily expands binary size. Some tentative experiments show that binary size can grow 5 to 6 times larger with respect to the original, if we perform the disassemble-reassemble iteration for 1,000 times.

We cannot identify the prologue and epilogue functions in COTS binaries as the symbol information has been stripped. However, after the first disassemble-reassemble attempt, we get an unstripped binary with sufficient information indicating which functions are added by the linker. If we are to do another disassemble-reassemble round, UROBOROS can skip these functions in the disassembly phase.

Another source of redundancy is the padding bytes in data sections. In ELF binaries,there are three data sections (`.data`, `.rodata`, and `.bss`) that have padding bytes at the beginning. As these padding bytes are not used, we remove them from the recovered program before reassembling.

With the code and data redundancy trimmed, binary size expansion is reduced to almost zero, no matter how many times a binary is disassembled and reassembled.

5.7 Main Function Identification

In a compiler-produced object file, the symbol information of the `main` function is exported so that it can be accessed by the libc prologue functions in the linking process. However, as this symbol information in executable file is stripped in COTS binaries after linking, we need to recover and export it before reassembling.

Through our investigation, we found that the code sequence shown below is typically used to pass the starting address of `main` to libc prologue function `libc_start_main`.

```
push $0x80483b4
call 80482f0 <__libc_start_main@plt>
hlt
```

The first argument of `libc_start_main`, which is `0x80483b4` in this example, is recognized as the starting address of the `main` function. We insert a label named `main` and the type macro `.globl main` in the output at this address.

5.8 Interface to External Transformation

As briefly discussed in §1, existing binary software security applications mainly rely on patch-based or instrumentation-based binary manipulations. We argue that given the assembly program and support for program-wide transformation from UROBOROS, we can bridge external instrumentation and analysis techniques with binary retrofitting application development. The program-wide security instrumentation such as CFI, ROP attack mitigation, randomization and software diversification could be ported on the basis of UROBOROS to legacy binaries, without the inefficiency, cumbersomeness and restriction brought by previous binary manipulation methods.

In order to demonstrate that UROBOROS is an enabling tool that makes analysis and transformations applicable to legacy binaries in general, we implement a diversification transformation based on basic block reordering. After disassembly, we walk through each function and randomly select two basic blocks from its CFG as the reordering targets. Control-flow transfer instructions and labels are inserted in the selected blocks, their predecessors, and successors to guarantee semantic equivalence. We perform this reordering *iteratively*, namely the output of each iteration becomes the input of the next round. We conducted a quick experiment on `gzip`. The disassembly-transformation-reassembly process was iterated 1,000 times. The effectiveness of the diversification transformation is evaluated by the elimination rate of ROP gadgets measured by the ROP gadget detector ROPGadget [40]. From this preliminary study, we find that it is much easier than binary rewriting to perform binary-based software retrofitting based on UROBOROS. As the ROP defense is not the focus of this research, we omit the detailed results in this paper.

Table 1: Programs used in UROBOROS evaluation

Collection	Size	Content
COREUTILS	103	GNU Core Utilities
REAL	7	bc, ctags, gzip, mongoose, nweb, oftpd, thttpd
SPEC	12	C programs in SPEC2006

6 Evaluation

We evaluate UROBOROS with respect to correctness, cost, and its ability to support program-wide transformation. The correctness verification examines whether UROBOROS's reassembly is semantic preserving. Evaluation on the cost of UROBOROS reveals its reassembly's impact on binary size and execution speed, and also the running time of UROBOROS itself. As presented in §5.8, we study UROBOROS's support for binary-based software retrofitting, by implementing a basic block reordering algorithm to diversify disassembled binaries and eliminate ROP gadgets. As we have emphasized, UROBOROS is an enabling tool for other security hardening techniques. However, as goal-driven software security hardening is out of the scope of this paper, we do not present the detailed experiment results here.

We use three collections of binaries compiled from C code to evaluate UROBOROS. The first set, referred to as COREUTILS, is the entire GNU core utilities including 103 utility programs for file, shell, and text manipulation. The second set, called REAL, consists of 7 real-world programs picked by us, covering multiple categories such as floating-point and network programs. The last set subsumes all the C programs in the SPEC2006 benchmark suit, thus will be denoted by SPEC. Details of each collection are listed in Table 1. In the evaluation we compile all programs for both 32-bit and 64-bit targets. Since there are 122 programs, the number of tested binaries is 244 in total. The compiler is GCC 4.6.3, using the default configuration and optimization level of each program. All experiments are undertaken on Ubuntu 12.04. For each test case, we use the `strip` tool from GNU Binutils to strip off the symbol information and debug information before testing.

Table 2: Functionality test input for REAL

Program	Test Input
bc	Test cases shipped with the program
gzip	Test cases shipped with the program
ctags	Parse a C source file of 152,270 lines
oftpd	Login and fetch a large file
thttpd	Request some web pages & a large file
mongoose	Request some web pages & a large file
nweb	Request some web pages & a large file

6.1 Correctness

We verify the correctness of UROBOROS's reassembility in two steps. First, we execute binaries assembled from UROBOROS's output with test input shipped with the software. Both COREUTILS and SPEC have test cases shipped with the software by default. As for the REAL programs, most of them do not have test cases, so we develop input by ourselves to verify the major functionality. The input we use for testing the REAL collection is listed in Table 2.

Second, we examine the false positives and false negatives of our symbolization process for all the binaries of the three collections. In our context, a false positive is an immediate value that we mistakenly symbolize, while a false negative is a symbol reference that we fail to identify.

As described in §4, we have different assumptions to guide the symbolization process, so the correctness of different assumption combinations are verified. Since the three assumptions are orthogonal, there are eight different combinations with the choices of the three assumptions. With limited resources, it is difficult to test all 244 programs on all assumption sets. With some tentative experiments, we found that A1 is an assumption which greatly improves the overall performance of our disassembly and reassembly method. Therefore, we reduce the eight candidates to five by always including A1 except in the empty assumption set. In detail, the five assumption sets applied are {} (empty set), {A1}, {A1, A2}, {A1, A3}, and {A1, A2, A3}.

For all tested assumption sets, all reassembled binaries from COREUTILS and REAL pass the functionality tests. Some binaries from SPEC, however, fail to pass the tests, which are listed in Table 3. With the assumption set {A1, A2, A3}, only the 32-bit version of `gobmk` from SPEC (out of 244 cases in total) fails the functionality test. By inspecting this defected binary, we successfully locate the cause of failure. Some 4-byte sequences in the data sections happen to contain the same value as the starting address of a function, but they are not code pointers. UROBOROS incorrectly symbolizes them, leading to false positives. After we correct these errors, `gobmk` successfully passes the test.

For symbol-level correctness verification, we provide the statistics on false positives and false negatives of symbolization. A false positive is an immediate value that should not have been symbolized. A false negative is an immediate value which should be symbolized but failed to be after our symbolization process. We obtain the ground truth by parsing the relocation information provided by the linker.

We have verified all binaries in this step. Due to limited space, we only list the results for non-trivial cases, namely programs with at least one symbolization false positive or false negative with any assumption combination. Table 4 and 5 show the false positive and false negative analysis for 32-bit binaries, and Table 6 reports false positive analysis for 64-bit binaries. There are no false negatives on any of the 64-bit binaries. We emphasize in particular that, *with {A1, A2, A3} applied, among all the 244 binaries, only `gobmk` has a few false positives, and none has false negatives.*

The results of symbol-level verification are highly synchronized with the results from the first stage—binaries reassembled with no false positives or false negatives can pass all test cases. The results show that symbolization errors are found in `gobmk` no matter which assumption set we apply. In particular, we have verified that symbolization errors found in `gobmk` when applying {A1, A2, A3} are all caused by program data colliding with some function starting addresses. These collisions cause a functionality test failure for 32-bit `gobmk`, but the 64-bit version can pass the test due to the incompleteness of test input. In summary, the two stages of verification together imply that all three assumptions proposed for symbolization are reasonable.

Although the symbolization errors occurring in the case of `gobmk` seem conceptually "general", our study shows that the collisions are actually rare in practice, unless the disassembled binary has very large data sections like `gobmk` does. See Appendix A for the symbolization errors in `gobmk`. On the other hand, UROBOROS can successfully disassemble large and complicated binaries like `gcc` and `perlbench`. Overall, the results from two stages of correctness verification suggest that UROBOROS is a promising tool with remarkable practical value.

6.2 Cost

The cost of UROBOROS manifests from three aspects: size expansion of reassembled binaries, execution overhead of reassembled binaries, and the processing time of UROBOROS itself.Due to space restrictions, we only report the evaluation results on 32-bit binaries in this paper.

Table 3: Dynamic test results on reassembled binaries

Assumption Set	Binaries Failing Functionality Tests	
	32-bit	64-bit
{}	h264ref, gcc, gobmk, hmmer	perlbench, gcc, gobmk, hmmer, sjeng, h264ref, lbm, sphinx3
{A1}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A2}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A3}	gobmk	gcc, gobmk
{A1, A2, A3}	gobmk	

Table 4: Symbolization false positives of 32-bit SPEC, REAL and COREUTILS (Others have zero false positive)

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	1	0.049‰	1	0.049‰	0	0.000‰	0	0.000‰
gcc	262698	49	0.187‰	32	0.122‰	32	0.122‰	0	0.000‰	0	0.000‰
gobmk	65244	1348	20.661‰	985	15.097‰	912	13.978‰	78	1.196‰	5	0.077‰

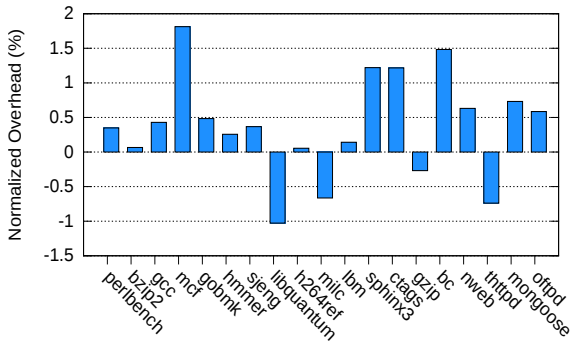

Figure 5: Execution overhead for REAL and SPEC programs relative to the original versions

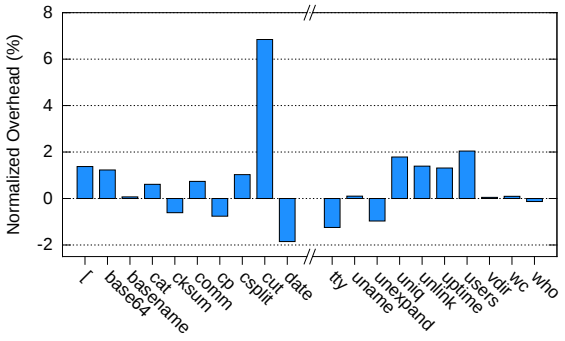

Figure 6: Execution overhead for COREUTILS programs relative to the original versions

6.2.1 Execution Overhead

Some programs in COREUTILS are not suitable for performance benchmarking, including `su`, `nohup`, and `timeout`, etc. After excluding these programs, we have 90 left to inspect in COREUTILS. The experiments are conducted on a machine with Intel Core i7-3770 3.40GHz and 8GB memory running Ubuntu 12.04.

We present the execution slowdown of reassembled binaries in Figure 5 and Figure 6. Since it is hard to present the data of all 90 binaries from COREUTILS, we sort COREUTILS programs by their names in alphabet order and plot the data for the first and last 10 programs in Figure 6. We report that the average slowdown for is 0.44% for COREUTILS, 0.29% for SPEC and 0.52% for REAL. The data suggests that UROBOROS does not have any significant impact on the execution speed of reassembled binaries.

6.2.2 Size Expansion

We use the `stat` program from GNU Coreutils to calculate file size expansion of the reassembled binaries compared to the originals. As the increase is generally negligible, we only report the average data here. The average expansion for COREUTILS is 0.83%, 0.00% for SPEC and -0.02% for REAL. Data shows that UROBOROS has almost zero impact on binary size when delivering reassembility. As aforementioned in §5.6, subsequent disassembly-reassembly iterations have zero expansion.

6.2.3 Processing Time

We measure how long it takes UROBOROS to disassemble binaries. Figure 7 presents the processing time for SPEC and REAL binaries. Figure 8 presents processing time for COREUTILS binaries selected using a same al-

Table 5: Symbolization false negatives of 32-bit SPEC, REAL and COREUTILS (Others have zero false negative)

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gcc	262698	11	0.042‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gobmk	65244	86	1.318‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰

Table 6: Symbolization false positives of 64-bit SPEC, REAL and COREUTILS (Others have zero false positive). Also, no false negatives are found for any binary.

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate
perlbench	76952	32	0.416‰	10	0.130‰	10	0.130‰	0	0.000‰	0	0.000‰
gcc	259213	506	1.952‰	126	0.486‰	14	0.054‰	112	0.432‰	0	0.000‰
gobmk	65255	2437	37.346‰	1079	16.535‰	7	0.107‰	1073	16.443‰	1	0.015‰
hmmer	13165	11	0.836‰	2	0.152‰	0	0.000‰	2	0.152‰	0	0.000‰
sjeng	8837	22	2.490‰	2	0.226‰	0	0.000‰	2	0.226‰	0	0.000‰
h264ref	20264	15	0.740‰	1	0.049‰	0	0.000‰	1	0.049‰	0	0.000‰
lbm	248	1	4.032‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
sphinx3	8656	3	0.347‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
ctags	12997	2	0.154‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gzip	3323	11	3.310‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
mongoose	3643	1	0.275‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
df	4202	1	0.238‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
du	4593	1	0.218‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
split	2851	1	0.351‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
timeout	1935	1	0.517‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰

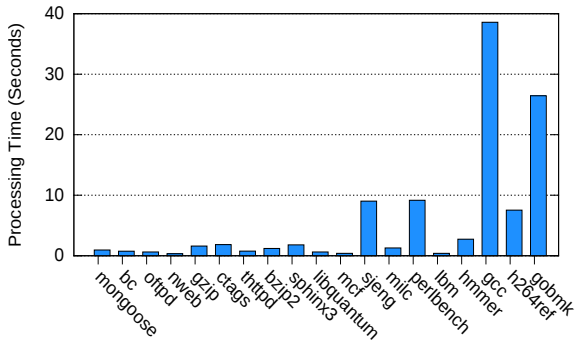

Figure 7: Processing time for SPEC and REAL binaries

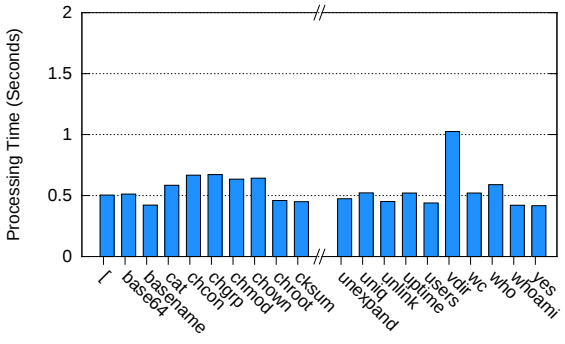

Figure 8: Processing time for COREUTILS binaries

phabet order strategy. As expected, larger binaries take more time to process. On average, UROBOROS spends 8.27 seconds on binaries from SPEC, 0.98 seconds on binaries from REAL, and 0.57 seconds on binaries from COREUTILS. We interpret this as a promising result, and the efficiency of UROBOROS makes it a tool totally practical for production deployment.

7 Discussions and Limitations

Compiler Compatibility. Sometimes binary reverse engineering is compiler dependent, but UROBOROS does not explicitly depend on any compiler-specific features as far as we know. To roughly investigate UROBOROS's compatibility with other compilers, we try to disassemble and reassemble some binaries compiled by Clang,

another widely used compiler different from GCC. We only briefly present the results here due to limited space. We repeat the same functionality verification described in §6.1 on the 32-bit binaries in REAL, which are compiled by Clang this time. The applied assumption set in this experiment is the empty assumption set, and all reassembled binaries can pass the functionality tests. We plan to test UROBOROS's compatibility in more depth in the future.

C++ Binary Disassembly. The C++ programming language has more specific features compared with C. Binaries compiled from C++ programs often contain more sections to store meta-information. At this point UROBOROS still cannot fully support C++ disassembly, but we have already worked out a blueprint on how to recover these sections. There are two kinds of meta-information sections specific to C++. We now briefly discuss how to recover them.

- The `.ctors` and `.init_array` sections contain the addresses of constructor functions—functions that need to be executed at start up before the `main` function takes control. These sections can be directly dumped out and symbolized by treating them as data sections.
- The `.eh_frame` and `.gcc_except_table` sections store the information used for stack unwinding and exception handling for C++ programs in the DWARF format [15]. There have been some reverse engineering tools, e.g., Katana [38] and IDA Pro, that can parse the DWARF data. By understanding the semantics of a DWARF entry, we can adjust its content and make the reassembly flawless.

We leave fully supporting C++ binary disassembly as part of our future work.

Availability of function starting addresses. We assume the availability of the function starting addresses in the input binary, as in this research we would like to assess the assumptions and techniques we develop for the symbolization problem. Identifying function starting addresses is an orthogonal research issue which has been the focus of recent work [6, 41]. UROBOROS can leverage existing techniques to make the tool more practical. Nevertheless, this is currently a limitation of UROBOROS and we plan to investigate further in the future.

Data Section Relocation. By accepting the assumption A2 (see §4), we fix the starting address of data sections, which leads to certain limitations related to the relocation of data sections. However, data can still be manipulated as long as the starting addresses stay the same. Besides, `.bss` section can be extended with new elements, as it is at the end of typical ELF binaries and the increase of its size does not need to relocate other sections. In the future, it would be interesting to see whether some more sophisticated heuristics or analysis can be developed to symbolize d2d references.

Extensions. We believe that we have built an enabling technology that could be employed as the basis of many important research and applications, such as software fault isolation (SFI), control-flow integrity (CFI), ROP defense, and in general software retrofitting for binary code, which is extremely important for legacy code systems. Nevertheless, this is a first step in the toolchain development. We plan to build and maintain a sustainable ecosystem, and also link to the existing ecosystems such as LLVM [27] and CIL [36] by lifting assembly to LLVM and CIL IR.

8 Conclusion

We have presented UROBOROS, a tool that can disassemble stripped binaries and produce *reassembleable* assembly code in a fully automated manner. We call this technique *reassembleable disassembling* and have developed a prototype called UROBOROS. Our experiments show that reassembled programs incur negligible execution overhead, and thus UROBOROS can be potentially used as a foundation for binary-based software retrofitting.

9 Acknowledgments

We thank the USENIX Security anonymous reviewers and Stephen McCamant for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grant N00014-13-1-0175.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security* (2005), ACM, pp. 340–353.

[2] ADL-TABATABAI, A.-R., LANGDALE, G., LUCCO, S., AND WAHBE, R. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (1996), ACM, pp. 127–136.

[3] ANAND, K., SMITHSON, M., ELWAZEER, K., KOTHA, A., GRUEN, J., GILES, N., AND BARUA, R. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 295–308.

[4] ANSEL, J., MARCHENKO, P., ERLINGSSON, U., TAYLOR, E., CHEN, B., SCHUFF, D. L., SEHR, D., BIFFLE, C. L., AND YEE, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2011), pp. 355–366.

[5] BALAKRISHNAN, G. *WYSINWYX: What You See is Not What You eXecute*. PhD thesis, University of Wisconsin-Madison, 2007.

[6] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. ByteWeight: Learning to recognize functions in binary code. In *In Proceedings of the 23rd USENIX Security Symposium* (2014), USENIX Association, pp. 845–860.

[7] BRUENING, D. L. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[8] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A binary analysis platform. In *Computer Aided Verification* (2011), Springer, pp. 463–469.

[9] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), ACM, pp. 27–38.

[10] BUCK, B., AND HOLLINGSWORTH, J. K. An API for runtime code patching. *Int. J. High Perform. Comput. Appl. 14*, 4 (2000), 317–329.

[11] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proceedings of the Network and Distributed System Security Symposium* (2010), The Internet Society.

[12] Dagger. http://dagger.repzret.org/.

[13] DE SUTTER, B., DE BUS, B., AND DE BOSSCHERE, K. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst. 27*, 5 (2005), 882–945.

[14] DENG, Z., ZHANG, X., AND XU, D. BISTRO: Binary component extraction and embedding for software security applications. In *Proceedings of 18th European Symposium on Research in Computer Security* (2013), Springer, pp. 200–218.

[15] DWARF debugging information format, 1993. Proposed Standard, UNIX International Programming Languages Special Interest Group.

[16] EDWARDS, A., VO, H., SRIVASTAVA, A., AND SRIVASTAVA, A. Vulcan binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research, 2001.

[17] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), USENIX Association, pp. 75–88.

[18] FORD, B., AND COX, R. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of USENIX 2008 Annual Technical Conference* (2008), USENIX Association, pp. 293–306.

[19] GRAHAM, S. L., LUCCO, S., AND WAHBE, R. Adaptable binary programs. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (1995), USENIX Association, pp. 26–26.

[20] HARRIS, L. C., AND MILLER, B. P. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News 33*, 5 (2005), 63–68.

[21] Hex-Rays Decompiler: Manual. https://www.hex-rays.com/products/decompiler/manual/failures.shtml.

[22] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 571–585.

[23] HORSPOOL, R. N., AND MAROVAC, N. An approach to the problem of detranslation of computer programs. *Comput. J. 23*, 3 (1980), 223–229.

[24] The IDA Pro disassembler and debugger. https://www.hex-rays.com/idapro.

[25] Introduction to x64 Assembly. https://software.intel.com/en-us/articles/introduction-to-x64-assembly/.

[26] KOLBITSCH, C., HOLZ, T., KRUEGEL, C., AND KIRDA, E. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 29–44.

[27] LATTNER, C. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2005.

[28] LEE, J., AVGERINOS, T., AND BRUMLEY, D. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium* (2011), The Internet Society.

[29] LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems* (2010), ACM, pp. 195–208.

[30] LLVM 3.1 release notes. http://llvm.org/releases/3.1/docs/ReleaseNotes.html, 2012.

[31] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), ACM, pp. 190–200.

[32] MC-Semantics. https://github.com/trailofbits/mcsema.

[33] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium* (2006), USENIX Association.

[34] MING, J., WU, D., XIAO, G., WANG, J., AND LIU, P. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium* (2015), USENIX Association.

[35] MUTH, R., DEBRAY, S. K., WATTERSON, S., AND DE BOSSCHERE, K. Alto: A link-time optimizer for the Compaq Alpha. *Softw. Pract. Exper. 31*, 1 (2001), 67–101.

[36] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), Springer, pp. 213–228.

[37] NIU, B., AND TAN, G. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, pp. 577–587.

[38] OAKLEY, J., AND BRATUS, S. Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code. In *Proceedings of the 5th USENIX Conference on Offensive Technologies* (2011), USENIX Association, pp. 11–11.

[39] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE.

[40] ROPgadget tool. http://shell-storm.org/project/ROPgadget.

[41] ROSENBLUM, N., ZHU, X., MILLER, B., AND HUNT, K. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2* (2008), AAAI, pp. 798–804.

[42] SCHWARTZ, E. J., LEE, J., WOO, M., AND BRUMLEY, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium* (2013), USENIX Association, pp. 353–368.

[43] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Conference on Security* (2010), USENIX Association, pp. 1–11.

[44] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security* (2007), ACM, pp. 552–561.

[45] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Berlin, Heidelberg, 2008), ICISS '08, Springer-Verlag, pp. 1–25.

[46] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev. 27*, 5 (1993), 203–216.

[47] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and Communications Security* (2012), ACM, pp. 157–168.

[48] ZENG, J., FU, Y., MILLER, K. A., LIN, Z., ZHANG, X., AND XU, D. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security* (2013), ACM, pp. 487–498.

[49] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 559–573.

[50] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proceedings of the 22Nd USENIX Conference on Security* (2013), USENIX Association, pp. 337–352.

A Symbolization Errors in gobmk

Among the 244 binaries from the COREUTILS, REAL, and SPEC collections, we found 5 d2c reference symbolization errors in `gobmk`, one of the largest SPEC programs, 4 of which are in the 32-bit version and 1 of which is in the 64-bit version, all false positives.

The software `gobmk` is GNU Go (http://www.gnu.org/software/gnugo/), a program for playing the board game of Go. The program contains a fairly large database of board configuration patterns. In order to speed up pattern matching, it builds Deterministic Finite Automata (DFA) from the pattern database.

The five reference symbolization errors are shown in Table 7 and Figure 9. In Figure 9, the two arrays `state_owl_attackpat` and `state_owl_defendpat` encode two DFAs with 24,701 and 34,044 entries, respectively. Each entry represents a state in the corresponding DFA. Each state has 5 numbers of the C short datatype, the current state and its four neighbors, as Go games are played on 2-dimensional grid boards.

We found two consecutive states in DFA `state_owl_attackpat` of the 64-bit `gobmk` are `{66,{0,0,0,0}},{70,{0,0,0,0}}`. The two C short int numbers 0 and 70 in the middle forms 0x460000 (little-endian), which happens to be the starting address of function `gtp_trymove`. Similar patterns exist in the 32-bit `gobmk`. States in array `state_owl_defendpat` forms value 0x080c0000 from two C short int numbers 0 and 2060 next to each other (little-endian), which collides with the starting address of function `autohelperpat1029`.

Table 7: Source code locations of symbolization errors

Program	Location (file and line no.)	
32-bit gobmk	`owl_defendpat.c:`	`9688`
	`owl_defendpat.c:`	`9702`
	`owl_defendpat.c:`	`9703`
	`owl_defendpat.c:`	`9704`
	`owl_defendpat.c:`	`9761`
64-bit gobmk	`owl_attackpat.c:`	`5828`

```
static const state_rt_t
    state_owl_defendpat[34044] = {
  ...
  {0,{2060,2061,2062,2063}}, ...
  {0,{2060,2060,2060,2060}}, ...
  {0,{2060,2060,2060,2060}}, ...
  {0,{2060,2060,2060,2060}}, ...
  {0,{2060,2060,2061,2060}}, ...
};
```

(a) 32-bit `gobmk`

```
static const state_rt_t
    state_owl_attackpat[24701] = {
  ...
  {66,{0,0,0,0}}, {70,{0,0,0,0}}, ...
};
```

(b) 64-bit `gobmk`

Figure 9: Source code of symbolization errors