Ramblr: Making Reassembly Great Again

Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry,
John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna
University of California, Santa Barbara
{*fish, yans, antoniob, machiry, jmg, pcgrosen, chris, vigna*}*@cs.ucsb.edu*

***Abstract*—Static binary rewriting has many important applications in reverse engineering, such as patching, code reuse, and instrumentation. Binary reassembling is an efficient solution for static binary rewriting. While there has been a proposed solution to the reassembly of binaries, an evaluation on a real-world binary dataset shows that it suffers from some problems that lead to breaking binaries. Those problems include incorrect symbolization of immediates, failure in identifying symbolizable constants, lack of pointer safety checks, and other issues. Failure in addressing those problems makes the existing approach unsuitable for real-world binaries, especially those compiled with optimizations enabled.**

In this paper, we present a new systematic approach for binary reassembling. Our new approach is implemented in a tool called `Ramblr`. We evaluate `Ramblr` on 106 real-world programs on Linux x86 and x86-64, and 143 programs collected from the Cyber Grand Challenge Qualification Event. All programs are compiled to binaries with a set of different compilation flags in order to cover as many real-world scenarios as possible. `Ramblr` successfully reassembles most of the binaries, which is an improvement over the state-of-the-art approach. It should be noted that our reassembling procedure yields no execution overhead and no size expansion.

I. Introduction

Our world is extremely software-dependent. Because of this, disruption caused by *flaws* in this software has significant impact in the "real world". These flaws come in two forms: bugs that simply affect functionality and bugs that lead to exploitable vulnerabilities. While the former cause their own level of havoc on our connected society, the latter are especially dangerous, since vulnerabilities can be leveraged by a proficient attacker to perform a larger-scale compromise. For example, an unpatched vulnerability in an internet-facing service could be exploited by attackers and used as a pivot point into the internal networks of the organization running the service. Because of this risk, patches to remediate exploitable bugs must be deployed as quickly as possible.

If the source code of an application is available, patching a bug is fairly straightforward: the source code is modified to preclude the vulnerability (e.g., by adding a safety check or refactoring application logic), and the program is recompiled. However, when source code is *absent*, such as in the case of proprietary software, the problem is much more complex. If the user of the software is unwilling to wait for the vendor to ship a new binary (or if the vendor no longer exists), the only option is to patch the binary directly.

Patching binary code introduces challenges not present when patching source code. When a patch is applied at the source code level, the compiler will redo the process of arranging code and data in memory and handling links between them. In binary code, this is extremely difficult, since this linkage information is discarded by the compiler once finished. A performant binary patching process would need to rediscover the semantic meanings of different regions of program memory, and *reassemble* the program, redoing the compiler's arrangement while preserving cross-references among code and data. As a result of the difficulties inherent to this procedure, the patching of binary code is currently an ad-hoc process. Current work in the research community either makes unrealistically strict assumptions, does not provide realistic functionality guarantees, or results in significant performance and/or memory overhead. Because of this, no tool currently exists that can automatically and reliably patch real-world binary software.

In this paper, we present a novel, systematic approach to the *reliable* patching of software. Our work builds on the *reassembleable disassembly* idea introduced by Uroboros [25], but eliminates many of its limitations, adds functionality guarantees (or, unlike prior work, the ability to abort the reassembly process when these guarantees cannot be met), and results in zero performance overhead compared to the original binary. We disassemble the original binary, properly identify symbols and intended jump targets, insert the necessary patches, and reassemble the assembly into the patched binary.

Our solution is based on advanced static analyses, which introduces moderate analysis time requirements. To accommodate situations in which quick patching is paramount (e.g., when an identified zero-day vulnerability needs to be patched as quickly as possible), we developed a series of workarounds that drastically reduce the analysis time requirements while relaxing some of the guarantees of functionality.

We describe our approach, discuss the workarounds, and evaluate our approach on two corpora of binaries: a large set of "realistic" binaries developed for the DARPA Cyber Grand Challenge, and the set of GNU Coreutils binaries that has also been evaluated by related work. Our evaluation measures the reliability of our binary rewriting approach and demonstrates an application in the form of the insertion of general binary hardening techniques into previously-unhardened binaries, and finds that we make *significant* improvements over the state of

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23225

the art. While existing work breaks between 15% and 60% of the binaries it rewrites, our approach results in a successful reassembly rate of over 98%.

We summarize our contributions as follows:

- We demonstrate that correctly disassembling and reassembling binaries on a large scale is not as easy as previous work has claimed. We identify several critical challenges in binary reassembling that are not thoroughly explored in previous work, and show that failing to tackle these challenges *will* result in broken binaries. Our solution eliminates several key assumptions made by previous work and greatly expands the scope of binaries that binary reassembling can be applied to.
- We propose a systematic solution, based on localized data flow analysis and value-set analysis, to solve these challenges in real-world binaries. Our solution allows a certain level of functionality guarantees, and allows a trade-off to be made between analysis speed and guaranteed functionality.
- With a new definition of procedures and an improved control flow graph recovery technique, our solution also makes it possible to freely rearrange functions when reassembling, which was never done by any previous work. This is critical in certain use cases.
- We implement our solution in a tool called `Ramblr`, and evaluate it on a large set of binaries from the DARPA Cyber Grand Challenge, as well as real-world binaries from GNU Coreutils. In order to capture important features of binaries that are mostly used in the real world, we amplify the amount of binaries in our dataset by compiling them with different optimization levels and compiler flags, further stressing our tool. To our knowledge, we demonstrate the first reassembling technique that works on optimized binaries. We also demonstrate several applications of binary reassembling, implement a few of them on top of reassembling, and evaluate the overhead compared with several alternative applications of binary patching, showing that our technique has a significantly lower execution overhead and higher functionality guarantees than the alternatives.

In the next section, we will provide an overview of related work and discuss how it relates to our approach.

II. Background and Related Work

Our technique builds on current work in the field to achieve safe reassembly of binaries. In this section, we provide an overview of the state of the art to provide a proper frame for our work.

A. Static Disassembling

There is much work in the literature regarding the correct and complete disassembly of binaries. *Linear sweeping* [7], which refers to sweeping from the beginning to the end of the executable region of a binary and decoding all encountered bytes as instructions, is the simplest such technique. The more advanced approach, that most disassembling techniques build upon, is *recursive traversal*. This technique starts from the entry point of a binary, resolves the targets of each control transfer, and recursively follows those targets to decode any encountered bytes [12]. Recent research on static disassembly mostly focuses on complicated corner cases in binaries [3], [15].

The de-facto standard in industry for binary disassembling is IDA Pro, although recently, other tools and systems, like Hopper, Binary Ninja, angr, BAP, Radare2, etc. have started to challenge its dominance [5], [21], [16], [23], [6].

Disassembly is the first step in control flow graph recovery, and other highly effective techniques have been developed to recover the control flow graph of a binary [12], [21]. Recent research suggests that, while modern disassemblers and disassembly techniques are able to achieve a high coverage of disassembled instructions on stripped, real-world binaries, properly identifying functions remains a challenge, especially on optimized binaries [1]. Even for the best techniques, accuracy falls drastically from 99%, on binaries compiled with no optimization (i.e., O0 optimization level in GCC), to only 82%, on binaries compiled with nearly full optimizations (i.e., O3 optimization level in GCC), combined with a noticeable increase in both false positives and false negatives [1]. As we will discuss in the following sections, current techniques have a pathological reliance on proper identification of function start points, a dependency that we remove with our approach.

B. Content Classification

After disassembling, the content within a binary must be *classified* (i.e., differentiated as either code or data) before the binary can be reassembled. This problem is formally referred to as *content classification*, and is believed to be difficult in binary analysis [22]. As previous research demonstrates, differentiating code and data statically is "unresolvable" in general, while doing so in a dynamic approach will inevitably face the classical problems of dynamic coverage and state explosion [11].

Recent work has been advancing the state of binary disassembling. While the problem is still unsolvable in general, we leverage, and improve, modern techniques stemming from the control-flow integrity (CFI) community for this purpose [29].

C. Binary Rewriting

Binary rewriting refers to the process of transforming one binary into another, either statically or dynamically, while maintaining existing functionality. Normally, one or more new features or behaviors are optionally added to the transformed binary during this process. *Static binary rewriting*, where the binary file itself is modified, generally introduces lower overhead when compared with dynamic counterparts, where binary code is instrumented *at runtime*. Thus, static rewriting is widely used in control flow integrity protection [29], [24], binary hardening [14], [27], security policy reinforcement [26], binary instrumentation, etc. Traditionally, static binary rewriting is either performed via *detouring*, which involves adding jump-out hooks to inserted code, or with full binary translation, lifting all code to an intermediate representation and translating it back to machine code. Both manners incur significant overhead on the resulting binary when compared with the original binary. In practice, full binary translation usually results in a binary that is very different, in terms of

cache locality and actual control flow, from the original one. Binary reassembling does not suffer from those drawbacks, as the reassembled binary is generated from the recovered assembly code, avoiding the need for detours or complete binary translation.

Dynamic binary rewriting techniques transform binaries as they are executing, and are able to guarantee a full-coverage transformation of commercial off-the-shelf (COTS) or stripped binaries at a high cost of performance overhead. Common dynamic rewriting tools include Pin, DynamoRIO [4], Valgrind [13], and Paradyn/Dyninst [10], which are all widely used in dynamic binary instrumentation.

D. Reassembleable Disassembling

```
804867d:    sub     esp,0x8
8048680:    push    DWORD PTR [ebp-0x10]
8048683:    push    0x80487c8
8048688:    call    80483e0
804868d:    add     esp,0x10
8048690:    sub     esp,0xc
8048693:    push    DWORD PTR [ebp-0x18]
8048696:    call    80483f0
804869b:    add     esp,0x10
804869e:    jmp     80486b9
```

Listing 1: An example output from objdump of a binary without relocation information.

The assembly generated by a disassembler is usually unfeasible for assembling, due to the lack of *relocation information*. Listing 1 shows a small piece of assembly extracted from a stripped binary. The reader can observe that the assembly contains absolute addresses, as opposed to labels. Theoretically, an assembler can take an assembly with absolute addresses and assemble it into a working binary. However, this approach is incompatible with binary patching and retrofitting, as it would require all of the basic blocks in the binary to be at their original positions. The crux of binary reassembling is the ability to relocate any binary code without any relocation information. The procedure that converts absolute addresses into corresponding labels, or symbol references, is called *symbolization*, which is the core of reassembling. Symbolization was first proposed and developed in *trace-oriented programming* [28], and then used by Uroboros to make reassembling binaries feasible [25]. Since our binary reassembling approach stems from the approach described in Uroboros, we first summarize the original approach here.

Symbolization attempts to determine whether an immediate value is directly or indirectly used as a symbol reference (i.e., whether it is *symbolizable*). In Uroboros, all references in a binary can be categorized into four different types, depending on the location of the reference itself and the location of its target: code-to-code (c2c), code-to-data (c2d), data-to-code (d2c), and data-to-data (d2d). Given predefined code and data memory regions from the binary, the symbolization process checks if the immediate value falls into any predefined memory region. If it does, a *symbol name* is created for the location and references to that location are changed from being absolute references (via immediates) to symbolic references (via the symbol name). Once this is done, code can be inserted into the re-symbolized assembly and the modified assembly can be reassembled into a new binary.

Uroboros makes many assumptions that preclude its use on many real-world binaries, and we propose significant improvements on its approach in this paper. In the next section, we will discuss Uroboros' limitations and why they prevent it from working on many binaries. After this, we describe our proposed approach, eliminating these limitations. In Section X, we evaluate our approach against the released implementation of Uroboros written and released by its authors and demonstrate improvements of our approach. Finally, in Section XI, we discuss limitations of our approach and give potential directions on future work.

III. Problems with Current Techniques

As discussed in Section II, our technique aims to achieve reassembly on a wider range of binaries than Uroboros does. Uroboros is a first step in the direction of reassembleable disassembly, but it does not perform well enough to work on many real-world binaries. This is due to simplifying assumptions made by the authors. We present their assumptions here, followed by a motivating example demonstrating failure cases for Uroboros, and a discussion of the challenging situations that cause such corner cases.

A. Uroboros' Assumptions

As discussed in Section II, Uroboros categorizes symbol references into four categories: code-to-code (c2c), code-to-data (c2d), data-to-code (d2c), and data-to-data (d2d). Because programs do not contain overlapping instructions, it is reasonable to assume that the symbol references that target code (c2c and d2c) must point to the beginning of an instruction [1]. To handle data-pointing symbol references (c2d and d2d) in Uroboros, the following three assumptions are made:

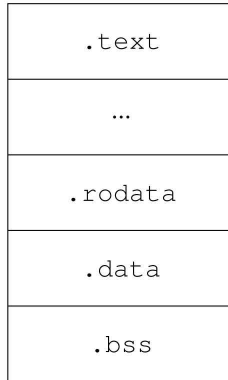
a) All pointers to the data region must be stored at an address aligned to the bit-width of the machine.
b) No transformation (i.e., of any base addresses) is required to be performed on the original binary. Hence in the reassembled binary, all data sections begin at the same address as their counterparts in the original binary.
c) d2c symbols are only used as function pointers or jump tables. Hence any d2c symbol reference must either point to the beginning of a function, or be part of an identified jump table.

With the three assumptions above, a very low false positive and false negative rate of symbolization is achieved in the original paper.

In the course of developing our approach, we identified cases of reassembled binaries being broken after applying the original symbolization approach. After investigation, we found that there are multiple complex corner cases that must be considered in order to symbolize all symbolizable immediates, and symbolize none of the non-symbolizable immediates. Further, we found that two of the original assumptions (assumption b) and c)) about d2c and d2d symbol references are too strict, which leads to the breaking of reassembled binaries, or do not support the goal of binary patching and retrofitting, which are important applications of reassembling.

[1] Specially constructed binaries could contain overlapping instructions, which both Uroboros and us ignore.

Fig. 1: A typical section layout of an ELF binary.



Assumption a) assumes that all pointers are stored at an aligned address. This assumption is generally acceptable, since most compilers *tend* to align pointers in memory for the sake of better performance. But it does not necessarily hold true for all data constructs. Listing 2 demonstrates one such example with a custom packing. A function pointer (field `cb`) is stored at offset 1 of struct `dp`. Assuming `dp` is stored at a machine bit-aligned address, the function pointer `my_callback` must be stored at an aligned address. Accepting assumption a) breaks any binary that has a data construct holding an unaligned pointer like Listing 2. Our technique supports unaligned storage and access of pointers, allowing us to handle arbitrary data structures.

```
typedef int (*callback)();

#pragma pack(1) /* Disables struct field aligning */
struct dp_t
{
    unsigned char flag;
    /* A function pointer stored at an unaligned
       address */
    callback cb;
};

static dp_t dp = {1, my_callback};
```

Listing 2: An example of unaligned pointer storage. Assume struct `dp` is stored at an aligned address 0x600000.

Assumption b) directly leads to the requirement that all data sections must be put at their original addresses in the original binary, which, in some cases, breaks binary retrofitting. Figure 1 shows a common section layout for ELF binaries on Linux. Usually, a binary may have a read-only section `.rodata`, a read-write section `.data`, and a section `.bss` which is initialized to all zeros at program start. If all three sections are close enough to each other, we cannot add any custom data to `.rodata` or `.data` section. Alternatively, a new section must be created to hold the newly inserted data, which, in certain cases where memory usage is a concern, is suboptimal. Our technique uses advanced static analysis to support arbitrary relocation and resizing of all sections, discarding this assumption in the process.

Assumption c) simply does not hold true in many binaries. The root issue is, as we mentioned in Section II-A, that identifying function start points on stripped binaries is still an unsolved problem, especially on binaries compiled with optimization enabled and C++ binaries. Hence relying on all function start points being successfully identified is not practical, which is why our solution does not assume a set of perfectly-identified function start points.

B. Motivating Failure Case

The successful operation of Uroboros depends on the perfect execution of its symbolization step. Uroboros linearly scans the data section of binaries, and considers every word-sized buffer whose integer value falls in a memory region to be a symbolizable integer. The idea seems straightforward, but it does not necessarily work due to false positives and false negatives during the symbolization step. An immediate might be a *symbolizable immediate*, meaning that it *should* be treated as a reference, or a *non-symbolizable immediate*, meaning that it might *look* like a reference, but is actually a true immediate. Any incorrect classification of the above during reassembling directly leads to the generation of broken reassembled binaries. As we will show throughout the rest of this section and in our evaluation, this incorrect classification happens rather frequently.

Uroboros depends on the three assumptions described previously in order to lower the chance of collisions between normal data and pointers, without the need of any advanced static analysis. Now we demonstrate why this approach is not generally applicable.

Consider the code snippet shown in Listing 3. Suppose the compiled binary has a `.text` section ranging from 0x8000000 to 0x8050000, a floating point variable `a` has an initial value of 4e−34. Its binary representation, as shown in the listing, happens to be `3d ec 04 08`, which is 0x804ec3d on little-endian machines. Uroboros mistreats the initial value as a symbolizable integer and symbolizes it. This is incorrect, and the binary is consequently broken after reassembling. The root cause is that Uroboros does not know the real type of any piece of data in the binary: all of its assumptions depend on the low odds of misidentification of a normal piece of data as a pointer.

```
static float a = 4e-34;
```

(a) Definition of a float variable `a`.

```
8060080     3d ec 04 08
```

(b) Binary representation of float `a`.

Listing 3: An example of a pointer value collision occurring in a float. `.text` section begins at 0x8000000 with size 0x50000.

As we discuss in this paper, `Ramblr` takes a different route of performing data identification and type recognition on the target binary to support the symbolization step. The more non-symbolizable data we identify, the less symbolization false positives there will be. By utilizing data identification and type recognition, our solution is able to identify the consecutive 4 bytes located at 0x8060080 as a floating point constant, and avoids symbolizing it.

In the following, we will detail the different challenges that Uroboros is unable to overcome, and that `Ramblr` addresses with advanced static analysis.

C. Unsurmounted Challenges

Many challenges arise when applying current reassembly approaches on a large set of real-world binaries. Like the motivating failure case in Section III-B, these challenges stem from corner cases that cause incorrect symbolization classifications. These classifications fall into two categories:

Symbolization false negatives. A symbolization false negative occurs when an immediate value (which is deemed as a non-symbolizable immediate initially) does not fall into any known memory region, but is used as part of a pointer in the binary.

Symbolization false positives. An immediate value, initially deemed as a symbolizable immediate, is sometimes simply a normal piece of data, causing a symbolization false positive, as shown above in our motivating failure case.

There are several categories of situations that cause symbolization mis-classifications. Here, we enumerate these categories with concrete examples.

Compiler optimizations. Due to compiler optimization techniques (namely, constant propagation and constant folding), a constant may be added to or subtracted from a pointer, creating a pointer to a different value. The target of this new pointer might appear to point outside of any memory region (causing a symbolization false negative) or to another memory region altogether (causing an incorrect symbolization).

Listing 4 is a xorshift pseudo-random number generator (PRNG) adapted from CGC binary `CROMU_00042`. We assume that the `state` array is stored at `0x80609e8`. According to the source code, variable `i`, which is the index counter of the loop, should range from 0 to 16, and memory addresses of the array assignment should be ranging from `0x80609e8` to $\texttt{0x80609e8} + 16 \times 8$. However, in the assembly compiled using Clang under optimization level `O1`, due to compiler optimization, the index variable `i` takes an initial value of $-\texttt{0x80}$ (which is -16×8), and the base pointer at instruction `0x804a33d` is `0x8060a68`, which is essentially $\texttt{0x80609e8} - \texttt{0x80}$. Uroboros cannot detect this occurrence. `Ramblr`, instead, addresses it by using *base pointer reattribution*, described in Section VII-A.

Abnormal binary behavior. In this case, the binary exhibits abnormal behavior (for instance, pointer encryption and decryption). If pointers are stored in a binary in a modified form, they might cause symbolization false negatives.

Adapted from CGC binary `KPRCA_00044`, Listing 5 shows an example of decryption of a jump target stored in `ecx` before using it as the target for call. The target function being called might be offsetted after reassembling, but since Uroboros cannot determine that the variable `encrypted_func_ptr` is the encrypted pointer of the target function, the pointer will not be symbolized, which results in a broken binary.

Since such binaries are rare in practice, and there is no generic way to handle those cases, we deem those binaries as unsafe for reassembling, and refrain from reassembling them. However, it is necessary to *detect* these cases to be able to opt out of reassembling. Uroboros has no functionality to handle these cases, leading to broken data references in the resulting binaries. `Ramblr` addresses this through its *data consumer check*, presented in Section VII-B.

```
/* Assume the array is stored at 0x80609e8 */
uint64_t state[16] = {0};

void sprng(uint64_t seed)
{
    uint64_t state_64 = seed;
    for (int i = 0; i < 16; i++)
    {
        state_64 ^= state_64 >> 27;
        state_64 ^= state_64 >> 13;
        state_64 ^= state_64 >> 46;
        state[i] = state_64 * 1865811235122147685;
    }
}
```

(a) An implementation of xorshift PRNG.

```
.text
...
; initial value of i is -0x80
804a2e4     mov    esi, -0x80
804a300     mov    eax, ebx
...
; beginning of the loop
804a300     mov    eax, ebx
...
; state[i] = state_64 * 1865811235122147685
804a32d     imul   ebp, ecx, 0x19071d96
804a333     add    ebp, edx
804a335     imul   edx, ebx, 0xd81ecd35
804a33b     add    edx, ebp
; write results to the state array
; note that 0x8060a68 - 0x80 = 0x80609e8
804a33d     mov    dword ptr [esi+0x8060a68], eax
804a343     mov    dword ptr [esi+0x8060a6c], edx
804a349     add    esi, 8
804a34c     jnz    0x804a300 ; loop end

.bss
...
80609e8     uint64_t state[16];
```

(b) An extract of the compiled PRNG in Clang with `-O1`.

Listing 4: An example where the base pointer appear to point outside of any memory region due to compiler optimization.

```
80480bb     mov    ecx, OFFSET FLAT: encrypted_func_ptr
80480c1     sub    ecx, OFFSET FLAT: encryption_key
; parameter to the function
80480c7     mov    dword ptr [esp], 0xDEADBEEF
80480ce     call   ecx
```

Listing 5: An example of pointer decryption before using the pointer as a jump target.

Value collisions. A frequent cause of broken reassembled binaries is value collision within the binary: a non-pointer integer happens to have a value that coincides with a location in a pre-defined memory region. This causes symbolization false positives, in which the colliding immediate is incorrectly symbolized, and its final value is wrongly modified in the reassembly process. Contrary to the argument in [25] that such collisions are “rare”, we find multiple cases in our dataset.

When reassembling more binaries, especially those compiled with optimization enabled, value collisions are not as rare as Uroboros claimed. For instance, Listing 6 shows a

simple collision we found in Coreutils program `factor` in byte array `primes_diff`. This array is the same (and with the same alignment) when compiled with different flags and optimization levels, but when compiled with `-O0`, `-O1`, and `-O2`, those binaries are not big enough, and as a result, the address 0x8060406 is not covered by any section in those binaries. A similar issue is found in Coreutils' ubiquitus program `ls`. Without handling such cases, the binary is broken by reassembling.

```
static const unsigned char
  primes_diff[] = {
    1, 2, 2, 4,
    2, 4, 2, 4, ...
    /* at 0x805da50 */
    2, 6, 4, 2
    /* at 0x805da54 */
    6, 4, 6, 8, ...
};
```

(a) Part of the `primes_diff` byte array.

```
.rodata:
805da50    .db 2
805da51    .db 6
805da52    .db 4
805da53    .db 2
805da54    .long 0x8060406
```

(b) Byte sequence at 0x805da54 falls into the memory region of this binary when decoded as a pointer.

Listing 6: An extract from `factor` compiled with GCC in `-O3` demonstrating a value collision occurred in array `primes_diff`.

A generic solution to this issue would require an analysis that can reason about the *purpose* of an immediate value in a binary. In general, this is not solvable, and Uroboros makes no attempt to compensate for this. `Ramblr` uses a set of best-effort approaches to mitigate this problem:

a) We perform a primitive data type recovery to identify the types of these data blocks. For example, if a 4-byte data block is recognized as a float constant, it should not be symbolized as a pointer. This is described in Section VI.

b) We perform an array size recovery to identify the size of some more complex program data constructs, like byte arrays, etc. For example, if a 128-byte data block is identified as a single byte array, none of the values inside should be symbolized as a pointer. This is also described in Section VI.

c) If an immediate value pointing to the middle of an instruction is first determined to be symbolizable, all previous decisions leading to this decision must be rolled back. Section VI-C contains details of this decision process.

Disassembly readability. Ideally, the disassembly file should be easy-to-read. Uroboros displays all non-symbolizable data in the form of individual bytes, which is very difficult for users to understand or edit. With the help of data type recognition, we are able to generate more natural-looking assemblies.

IV. Approach Overview

To make our technique more approachable, we present an overview of the technique in this section before describing it in-depth throughout the rest of the paper.

`Ramblr` works in several main steps when reassembling a target binary, each of which will be discussed in a subsequent section:

Disassembly and CFG Recovery. First, `Ramblr` recovers a complete CFG of the target binary, fully disassembling each basic block as it is identified. We discuss this in Section V.

Content Classification. Next, `Ramblr` classifies the contents of the target binary into several types (i.e., code, pointers, arrays, etc). `Ramblr` uses a combination of advanced static analysis techniques, along with metadata available in the target binary, to accomplish this task. Our classification process is described in Section VI.

Symbolization. Using the results of the previous two steps, `Ramblr` identifies symbol references in the target binary. These references identify the semantic meaning of a memory location (i.e., "the start of function X"), as opposed to the syntactic meaning of the address (i.e., "this code is at address Y") and are used in the reassembly step to maintain relationships from a reference to the object it points to. Symbolization, with our various improvements over previous work, is presented in Section VII.

Reassembly. With the symbols identified, *reassembleable* assembly code is generated for the binary. Any desired modifications to the binary are done on top of this assembly code – instructions can be added, removed, or replaced, and functions or data can be added. The modified assembly is then reassembled using an off-the-shelf assembler. The resulting binary is a functional application that exhibits the desired change of behavior from the original. We delve into this process in Section IX.

Throughout the rest of the paper, we will detail, discuss, and evaluate the steps summarized above.

The *content classification* and *symbolization* steps require static analyses that have moderate runtime requirements. To address the case in which reassembly must happen extremely quickly, we have developed a set of workarounds that increase the speed of our technique at the cost of some functionality guarantees of the resulting binary. We present these workarounds in Section VIII.

Despite our advancements in the technique of reassembling binaries, there are still cases where `Ramblr` cannot guarantee the functionality of the resulting binary. In these cases, it will emit an error message and refuse to reassemble the binary.

V. CFG Recovery and Disassembly

Before a target binary can be reassembled, it must be disassembled. We do this by computing a control flow graph (CFG) of the target binary and disassembling any identified basic block. Aside from this, we attempt to identify and disassemble dead code, as it is important for our approach that as much of the code as possible is disassembled.

We use the `angr` binary analysis framework for CFG recovery. If `angr`'s CFG recovery fails on the target binary, `Ramblr` is unable to continue and reports an error message. However, we did not find such cases in our test dataset.

We will briefly summarize how `angr`'s CFG recovery works and the slight modifications that we made to it to improve the disassembly coverage. While we summarize the approach in this section, we encourage the interested reader

to explore the angr authors' full description on the design of their CFG recovery [21].

A. Recursive CFG Recovery

CFG recovery starts from the entry point of the binary, and recursively follows direct control flow transitions or resolves and follows indirect control flow transitions. Eventually, the recovery exhausts the recursively reachable basic blocks of the executable regions of the binary (typically the .text section for ELF binaries), and disassembles as many bytes as possible.

B. Utilizing Meta Information

angr respects certain meta information from the binary, which includes segment and/or section information[2]. angr's CFG recovery assumes that non-executable memory regions only contain data bytes, not executable code.

C. Iterative Feedback

While the CFG recover and the latter steps are described separately in this paper, there is a backward flow of information in our implementation. If the later *Content Classification* step identifies a code reference (i.e., a hardcoded pointer in the data segment of the binary that points to the code segment), we inject it into angr as an additional target of a fake control flow transition, so that the recursive CFG recovery can explore that code block [3].

VI. Content Classification

To avoid the pitfalls discussed in Section III, we leverage advanced static analysis techniques to classify potential sources of references. We detail these techniques, and their application, in this section, and describe how they are ultimately used for symbolization in Section VII.

Our analyses add a reasonable runtime requirement to the reassembling process. In addition, they yield a certain level of functionality guarantee (discussed below), which makes it possible for reassembler to opt-out when facing binaries with bizarre features, rather than reassemble and break them. That being said, in cases when reassembly speed is absolutely critical, we have developed workarounds that avoid the runtime of the static analysis in exchange for a lower functionality guarantee of the resulting binary. Those are discussed in Section VIII.

Our approach to content classification uses two fundamental analyses:

Intra-function Data Dependence Analysis. We perform a blanket execution [8] on basic blocks in a specific function, from which we recover data dependencies between variable and constant definitions. Variable definitions include registers, stack variables, and memory cells. The scope of this execution is strictly confined to the current function. We assume all calls to other functions return an unconstrained value [4], as long as the callee returns.

Localized Value-set Analysis. Value-set analysis is first proposed as an abstract interpretation technique to statically analyze machine code [2], [18]. Instead of running value-set analysis on the entire binary, or a whole function, we designed a constrained version of value-set analysis, called *localized value-set analysis*, that only runs on a slice of the binary, such as a set of basic blocks, a loop, etc. With the result from data dependence analysis, we are able to build a program slice with respect to a memory access that acts on data representing potential mis-classifications of content. Running localized value-set analysis on the slice usually gives us enough information regarding the classification of the content used by the final memory access.

Unlike traditional static analyses, those two analyses used in our approach are heavily constrained and localized in order to make them fast and tractable. Empirically speaking, those localized analyses are generally sufficient for the use cases in our approach: resolving jump tables, recovering primitive data types, and retrieving the sizes of arrays accessed in simple loops. We use these analyses to support the data type recovery and the segregation of different blocks of data from each other. Both are used to reduce symbolization mis-classifications in the next step of the reassembly process.

A. Data Identification and Type Recognition

By analyzing a recovered CFG, some data in the binary can be identified and its type recognized, a procedure we call *data identification and type recognition*. Several approaches, including data dependence analysis, program slicing, and value-set analysis are integrated in our solution to recognize data types with a high identification rate. Data identification and type recognition, although not evaluated in this paper, is very useful in generating correct disassembly when inline data exists in the binary, as it avoids symbolization classification errors in the symbolization step.

Here we use jump table recovery as an example to demonstrate how this approach works: a local backward program slice is first generated with respect to the jump target, followed by the application of value-set analysis on the generated slice to recover the entries of the jump table and addresses of all possible jump targets. Once the entries of the jump table are recovered, we mark the range as data with a data type of "pointer array".

For binary reassembling, it is important to correctly differentiate symbolizable and non-symbolizable data, since symbolizing a non-symbolizable data entry or vice versa will lead to a broken resulting binary. Table I shows all types of data that Ramblr recognizes at the moment. They fall into several broad categories:

Primitives. This includes pointers, bytes, shorts, integers, floats, doubles, and so on. They are recognized by an-

[2]PE files do not have segments, while ELF files normally have both sections and segments, sections are not necessary for execution.

[3]Although linear sweeping will always find the code block eventually, finding the basic block as early as possible, and starting decoding the basic block at a correct location is still beneficial with respect to reducing the number of overlapping or incorrectly-started blocks caused by inline data or function alignments.

[4]The notion of *unconstrained values* is not employed in blanket execution. This can be seen as an abstract value that satisfies any comparisons.

[5]Depends on the bit-width of the binary.

Data type	Size	Symbolizable
pointer array	multiple of 4/8[5]	Yes
region boundary	0	Yes
DWORD	4	No
QWORD	8	No
32-bit floating point	4	No
64-bit floating point	8	No
80-bit floating point	10	No
128-bit floating point	16	No
null-terminated string	variable	No
non-null-terminated string	variable	No
null-terminated UTF-16 string	variable	No
non-null-terminated UTF-16 string	variable	No

TABLE I: All data types `Ramblr` currently recognizes.

alyzing instruction and data access patterns during CFG recovery and localized value-set analysis.

Strings. ASCII strings and Unicode strings. Identified by propagating types from known string manipulating functions (like `strlen`, `strcpy`, etc.) and scanning printable characters.

Jump tables. Jump tables, from indirect jump resolution.

Arrays of primitives. These are recognized by performing an intra-function data dependence analysis and localized value-set analysis.

B. Data Block Sanitization

All data blocks recognized from the previous step are *sanitized* to avoid overlapping data. The requirement is simple: any identified data block should not overlap with another identified data block. Data block overlapping arises when one data block is part of another data block or due to a misidentification occurring during data type recognition. The first case is common, and is easy to handle - `Ramblr` simply merges the two data blocks. Handling the second case is more difficult, as it is not always clear which data blocks are misidentified, or both of them are misidentified. We discuss ways to handle misidentification in Subsection VI-C.

C. Handling Misidentification

Misidentification of data blocks usually arises from the following scenarios:

A data block being accessed in multiple ways. A data block might be accessed in a different manner in different places. Consider Listing 7 as an example: the `personal_info` struct is accessed as a whole in function `zero_fill()`, and then each field of the struct is accessed individually later. During data identification and type recovery, multiple data blocks spanning `personal_info` are seen, and they have conflicting types: the one accessed from `zero_fill()` is a 12-byte "unknown" block, while the other one accessed from `initialize()` contains two integers and one pointer-array of length 1.

Failure in localized value-set analysis. Due to the fact that our localized value-set analysis runs on a slice of the program generated from a best-effort (and, thus, potentially incomplete) data dependence analysis, the value-set analysis might be processing incomplete code when recovering data sizes and types. Generally, a data block spanning from the beginning address to the maximum address (e.g., upper bound of the section it belongs to) is seen when such failures occur.

```
struct personal_info_t {
    unsigned char* name;
    unsigned int age;
} personal_info;

void zero_fill()
{
    /* memset() is inlined by the compiler. */
    memset(&personal_info, 0, sizeof(personal_info));
}

void initialize()
{
    personal_info.name = "Beatrice";
    personal_info.age = 25;
}

void rename(char* new_name)
{
    /* from the assembly, we can only say
     * personal_info.name is a four-byte integer. */
    personal_info.name = new_name;
}
```

Listing 7: An example of a data block being accessed at multiple locations in different ways.

The strategy we take to handle misidentification is two-fold. First, we prioritize smaller data blocks over larger ones. This is because, like in our example, programs tend to initialize data in bulk, then access individual fields in the *proper*, type-dependent manner. In the example of Listing 7, since the 12-byte block identified from `zero_fill()` is larger than other three blocks identified from `initialize()`, the latter ones are taken as identification result. Second, we prioritize symbolizable data types over non-symbolizable data types. This is because if a piece of data is *ever* accessed as a symbol by the program, then it *should* be treated in one during reassembly. Since `personal_info.name` is identified as an integer from `rename()` function and a pointer in `initialize()`, we correctly prioritize the symbolizable data type.

In many cases, static analyses we perform are not sufficient to find *all* data blocks used in binaries. Usually there are gaps between identified data blocks, in which case, `Ramblr` resorts to workarounds to find pointers inside, as described later in Section VIII. Note that the static analyses performed here significantly reduce the number of unsafe assumptions that `Ramblr` has to make during symbolization, which in turn reduces false positives.

VII. Symbolization

During the original linking process of the target binary, all labels in the object files are converted to absolute addresses. During reassembly, the location of the data and code in a target binary will likely change due to the modifications performed on it. If there are hard-coded pointer addresses or absolute jumps in the binary, they must be adjusted to target the new locations of the data or code to which they used to point. In fact, even relative jumps must be adjusted, as the insertion of code into or removal of code from basic blocks will change the offsets of basic blocks from one another.

The assembler can make these adjustments during the *reassembly* step (see Section IX), but it needs to be provided the information of what references reference which locations. To do this, we convert these references from hard-coded

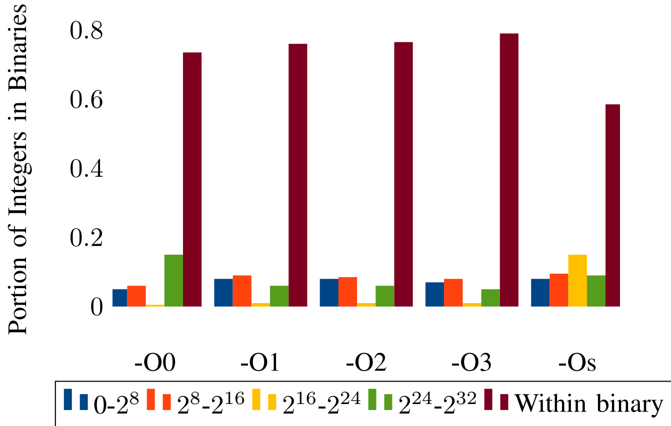


Fig. 2: Distribution of integers (instruction operands and data) in the 32-bit Coreutils and CGC datasets. (The first four buckets exclude integers within the binary.)

numerical references (absolute addresses or relative offsets) to *symbols*. This procedure, called *Symbolization*, converts absolute addresses back to labels, allowing relocation of the binary during the reassembly step.

However, not all immediate values should be converted to symbols, i.e., symbolized. We call all immediate values that must be symbolized in reassembling *symbolizable immediates*, and all other immediates (i.e., actual constants) *non-symbolizable immediates*. A successful binary reassembling requires that all symbolizable immediates are converted to correct symbols, and none of the non-symbolizable immediates are converted.

Symbolizable immediates exist in both code and data. In code, symbolizable immediates must be part of instructions, used as pointers pointing to either code or data. In data, symbolizable immediates are integers that are of machine's bit-width, which is also used as pointers pointing to either code or data.

Figure 2 illustrates a distribution of integers in an x86 binary. It is worth noting that most integers fall in the range between the beginning and the ending of memory regions of the binary, because those integers are used as code or data references. If all immediate values falling in the range or memory regions are symbolizable, and all immediate values falling outside are non-symbolizable, then symbolization is simply mapping those values within the range into symbol references. This is a common case, but it is not the general case. In Section III, we described several challenges that cause mis-classified symbols and result in broken reassembled binaries. In this section, we detail how `Ramblr` surmounts these challenges to properly handle binaries that current techniques fail to reasssemble.

A. Base Pointer Reattribution

Conceptually, a pointer is a reference to a memory location. At some point in the life of the program, it will be dereferenced so that value located at the memory location it references is retrieved, or used as a jump target. If a pointer is never dereferenced or used as a jump target anywhere in the binary, it is the same as an integer. For the purpose of reassembling, the ideal case is that all integers (including immediate operands or integers in data) in the binary can be categorized into two groups *solely* based on their values, which is described as *classification in symbolization* in Uroboros [25]. Their approach, in short, symbolizes integers to point to offsets in each memory region of the binary as long as the value of the integer falls into that region. If the integer does not fall into any memory region, it is marked as an integer, and will not be symbolized.

The original approach seems plausible. However, it does not always hold in real-world binaries, especially in binaries compiled with optimization enabled, due to constant propagation and constant folding performed during compiler optimization (as described in Section III. Consider the sample C code shown in Listing 8, and its assembly shown in Listing 9 compiled by `gcc` with flag `-O1`. For the ease of understanding, some unnecessary assembly lines are omitted, and the C code is put on top of each corresponding line of assembly.

```
int counters[2] = {0};

int main()
{
    int input;
    input = getchar();
    switch(input - 'A')
    {
        case 0:
            puts("option A");
            break;
        case 1:
            puts("option B");
            break;
        default:
            puts("Unknown option.");
            _exit(1);
    }
    counters[input - 'A'] ++;
}
```

Listing 8: An example of a base pointer pointing to outside of any memory region.

```
.text
; input = getchar();
80484ff    call    __IO_getc
8048504    mov     ebx, eax
; switch(input - 'A')
8048506    cmp     eax, 0x42
8048509    jz      short 0x8048523
...
; counters[input - 'A'] ++;
8048557    add     DWORD PTR 0x8049f30[ebx*4], 1

.bss
; int counters[2] = {0};
804a034    counters[0]
804a038    counters[1]
```

Listing 9: The assembly manifest of Listing 8, compiled by `gcc` with `-O1`.

The instruction at offset `0x8048557` increments the dword [`0x8049f30` + `ebx` × 4] by 1, where `ebx` holds the option letter (either "A" or "B") from user input. Due to optimizations, the pointer `0x8049f30` comes from address of the `counters` array (`0x804a034`), minus 4 × `0x41`, where `0x41` is the ASCII

code of character “A”. Since the integer value of this pointer does not fall into any memory region defined in this binary, it will be viewed as non-symbolizable by the original approach, and consequently the reassembled binary is functionally broken. The problem can be even worse: a symbolizable integer having, due to optimizations, a value inside the `.rodata` section might *actually* be pointing to the `.bss` section when being dereferenced. Such cases, which cause extremely hard-to-detect symbolization mis-classifications, are not rare in binaries compiled with optimization enabled.

To tackle this problem, we adopt a different approach. Instead of checking if the integer falls into any predefined memory regions of the binary, we enlarge each memory region by some amount, both in the beginning and the end, and check if the integer falls into any of the enlarged memory regions. This is used as a pre-filter to identify potential cases of mis-classification due to constant folding. Empirically, we enlarge each memory region by `4KB`.

For each symbolizable integer that matches our pre-filter, a forward slice is computed in the intra-function data dependence graph, until a dereferencing site of any value depending on the integer is reached. Then value-set analysis is performed on the slice, from the beginning to the dereferencing site, and an address (expressed as a value-set in VSA) is obtained. At this point, since the pointer must be valid when it is dereferenced, we can reasonably infer that the original symbolizable integer must point to the same memory region as this address belongs to. This approach not only makes it possible for `Ramblr` to correctly handle the example we described above (where the value of the pointer no longer falls within the bounds of the binary), but also finds and fixes cases where a base pointer points to one memory region, but in fact it should be pointing to another memory region in the binary.

B. Data Consumer Check

After previous steps, all immediates and constant values should be categorized into two groups: symbolizables and non-symbolizables. `Ramblr` is normally guaranteed to be correct as long as the above categorization is perfect. However, there are certain scenarios where categorization fails. Such scenarios are rarely seen in normal binaries, but arise when a binary implements unusual behavior, such as pointer decryption, custom pointer construction (e.g., adding two integers together, then converting the result to a pointer and dereferencing it), etc. We developed a *data consumer check* analysis that detects these scenarios in two ways:

1) For each non-symbolizable integer, data consumer check performs an intra-function data dependence analysis to determine if it is used as a pointer or a jump target later without involving any symbolizable integer. Specifically, the requirement to avoid involvement of any symbolizable integers excludes the *pointer offset* case from the *pointer construction* case. The former is already handled by making the base pointer properly symbolizable. The latter, on the other hand, results in a symbolization mis-classification and a broken binary. Intuitively, building a pointer out of integers, although acceptable, is an uncommon behavior, and we have found no way for it to be safely handled, in the general case, by binary reassembling.
2) For each symbolizable integer, the data consumer check performs an intra-function data dependence analysis on it and examines if any “unusual” operation is applied on it. Unusual operations include operations besides add and subtract (which are used for *pointer offsetting* and can be supported by reassembling). If hard-coded pointers undergo such operations, we assume that the binary is doing something unusual that reassembly cannot handle.

Reassembling immediately terminates when any of the cases above are found, as the reassembled binary would otherwise likely to be broken.

An example of pointer encryption is shown in Listing 10. A pointer in the binary is encrypted before use by XORing with a static number `0xdead1337`. It is loaded into register eax and decrypted before being used as a call target. Data consumer check recovers a data dependence graph with respect to the integer `0xdeed1137` loaded at instruction `0x400100`. This analysis detects that two non-symbolizable integers are XORed, and then used as a jump target. Data consumer check deems this binary to be unsafe for reassembling, and terminates reassembling right away.

It is important to note that, unlike prior techniques, `Ramblr` is able to detect these cases and avoid producing a broken reassembled binary.

```
.text
400100    mov   eax, DWORD PTR [0x600010]
400105    xor   eax, 0xdead1337
40010a    call  eax  ; calling address 0x400200

.data
600010    0xdeed1137
```

Listing 10: An example of pointer decryption using a static key.

VIII. Fast Workarounds

The systematic approach described in previous sections uses data dependence analysis and value-set analysis to offer a level of functionality guarantee for the reassembled binary. However, those analyses, along with the CFG recovery requirement, are still inevitably time-consuming on real-world binaries. In certain cases, where abundant test cases exist for the original binary and checking the functionality of the reassembled binary by running those test cases can be done quickly, some ad-hoc alternatives can be applied instead of the Content Classification and Symbolization steps of our systematic approach. This allows binary reassembling be done almost instantly, at the cost of some functionality guarantee.

This set of what we term “fast workarounds”, along with a discussion of their compromises on the functionality guarantees of the reassembled binary, are presented and discussed in this section. We measure the resulting correctness and the runtime of both the systematic approach and the fast workarounds in Section X.

A. Fast Data Type Recognition

In order to identify data types, especially to get the sizes of arrays, our systematic approach leverages localized static

analysis, which is accurate but heavyweight. An alternative approach is to *guess* data types based solely on the values of those data, which is way faster, and still maintains an acceptable accuracy for reassembling.

We implement a series of fast data type guessing strategies in `Ramblr`:

Floating point numbers. Our type guessing strategy for floating point numbers does a scan of the disassembly to identify obvious cases of data being used as floating points.

Pointer arrays. One or more consecutive integers of machine bit-width that points to any pre-defined memory region. We still apply a fast version of base pointer reattribution on pointer arrays, allowing for the identification of pointers that are, ostensibly, not pointing to any memory region due to compiler optimization. We treat individual pointers as a single-element pointer array.

Null-terminated Unicode strings. Any fully-printable consecutive sequence of valid Unicode characters, ending with *two* null bytes (by Unicode spec), is recognized as a null-terminated Unicode string. The minimal length is four characters.

Null-terminated ASCII strings. Any fully-printable consecutive byte sequence ending with a null byte is recognized as a null-terminated ASCII string. The minimal length is four bytes.

Sequences. Any arithmetic progression of bytes, shorts, or ints is recognized as a sequence. The minimal length is five elements.

Integers. We identify remaining "lone" integers by detecting integer-sized gaps in the remaining disassembly. This has no effect on the functionality of the reassembled binary, but it makes the disassembly more readable.

Unknown data. A linear sweep is performed on the entire non-executable memory region, and all gaps (bytes not belonging to any recognized data blocks) are identified as unknown data blocks.

Note that the order of applying these guessing strategies matters. For example, we cannot apply the "unknown data" identification strategy before other strategies are applied, otherwise all bytes will be identified as `unknown`. We apply these strategies in the order listed above.

`Ramblr`'s data guessing is easily extensible: users can add more type guessing strategies with respect to the nature of binaries to be reassembled, which will benefit the symbolization procedure by lowering potential misidentification of symbolizable immediates. If a binary embeds, for example, a PDF as a resource, a "PDF file" identification strategy can be easily added.

B. *Fast Base Pointer Reattribution*

As discussed previously in this paper, one issue that occurs during symbolization is that an immediate holding a value belonging to one memory region (or even outside any memory region) actually points to another memory region when dereferenced. This is generally caused by compiler optimizations. The issue is addressed by our *base pointer reattribution* in the systematic approach, involving intra-function data dependence tracking and value-set analysis. These analyses are both expensive. Given that an immediate being used as a pointer must be valid (i.e., must point to the appropriate memory region at dereferencing time), we perform a forced concrete execution on *any* path starting from the source of the immediate and ending at the dereferencing site that depends on the immediate value. Then, we symbolize the immediate value as an offset to the beginning of the memory region that the final dereference was targeting. For the sake of performance, we only process immediate values that are not trivially identifiable as belonging to any memory region.

The fast base pointer reattribution allows us to avoid *symbolization false negatives* by correctly detecting immediates as symbols in cases where they would otherwise be ignored.

IX. Reassembly

The reassembling procedure is straightforward. Taking results from symbolization, we first assign labels for every symbol reference we recovered, and then replace all symbolizable immediate values in each instruction and each data region with corresponding labels. The resulting reassembled disassembly is output into a single assembly file, to which the user can apply their own patches as needed. Finally, an off-the-shelf assembler is used to assemble the resulting assembly into a reassembled binary.

Theoretically, the assembly syntax can be either Intel or AT&T. `Ramblr` supports emitting either syntax, however, we find that Clang (from version 4.4 to the latest version 4.8) cannot support certain Intel-style instructions. Neither GCC nor Clang has issues supporting assembly in AT&T syntax. Therefore, `Ramblr` defaults to AT&T syntax. For the purpose of transparently supporting user patches written in a different syntax than the target outputting syntax, we also implement a syntax converter from Intel to AT&T style.

X. Implementation and Evaluation

This section covers the implementation of our prototype, `Ramblr`, describes the datasets that we use, and presents its evaluation. We evaluate the correctness of `Ramblr` against ground truth produced during original compilation of the binaries, compare it against Uroboros on two datasets, and discuss analysis time and execution overhead in the resulting binaries.

A. *Implementation Overview*

We use `angr`, an open-source binary analysis framework, as the platform for reassembling. `Ramblr` is implemented in Python as an `angr` analysis, and utilizes other publicly-available analyses routines in `angr`. All of our CFG recovery improvements are done on top of `angr`'s CFGFast analysis. `Capstone` is used for performing the disassembly of instructions [17]. Our prototype works on x86 and x86-64 ELF binaries. However, as `angr` is platform-independent, there are no fundamental limitations preventing an extension of `Ramblr` to other architectures. All of our evaluations are performed under PyPy 5.3.1 in Ubuntu Server 16.04 LTS.

The entire `Ramblr` toolchain, including `Ramblr` itself and our assembly syntax converters, is open sourced. `Ramblr`

Dataset	Total # of binaries	Optimization level	# of binaries
Coreutils	106	O0	106
		O1	106
		O2	106
		O3	106
		Ofast	106
		Os	106
CGC	143	O0	141
		O1	117
		O2	116
		O3	116
		Ofast	116
		Os	119

TABLE II: Number of binaries of each dataset.

is included in `angr`, while other parts of the toolchain are included in a binary patching platform called `Patcherex`, which was used by the third-place winning team in the DARPA Cyber Grand Challenge [20].

B. Dataset

We use two sets of binaries for the evaluation. The first set is `Coreutils` 8.25.55-ff217, which includes 106 different binaries that form much of the base of a Linux system. According to [25], `Coreutils` is one of the binaries collections used to evaluate Uroboros, and allows us to compare our approach to Uroboros. To test the relative versatility of the two approaches in the presence of advanced binary constructs, we compile each program in x86 and x64, with six different optimization levels, including `O0`, `O1`, `O2`, `O3`, `Ofast`, and `Os`, with GCC 5.4.1 in our testing environment.

The second dataset is a collection of 143 binaries from the Qualification Event (CQE) and the run-up to the Final Event (CFE) of the DARPA Cyber Grand Challenge (CGC), representing all CGC binaries released before August 2016. CGC binaries are stripped, self-contained x86 binaries that do not rely on any dynamically-linked libraries. We compile each program with six different optimization levels, including `O0`, `O1`, `O2`, `O3`, `Ofast`, and `Os`, with Clang 4.4 (the only supported compiler for CGC) in the DECREE VM provided by DARPA.

Note that some binaries in both datasets simply do not work (crashing with segmentation faults, failing test cases, etc.) when compiled with certain optimization levels. Those binaries are removed from each dataset. We also remove all multi-CB binaries from the CGC dataset as it is difficult to tell exactly which one of the full set of binaries is the culprit when a test case fails. The final count of binaries our datasets across different compilation flags is shown in Table II. The entire dataset is available upon inquiry.

Test Cases. Both Coreutils and CGC binaries come with abundant test cases, making them well-suited for evaluating the functionality of reassembled binaries. We run test cases on every reassembled binary, and mark a binary as broken if any test case fails.

C. Pre-evaluation

The authors of Uroboros [25] open-sourced their prototype implementation to the community [19]. We used their code for evaluating their approach in our comparative evaluation. However, we had to make several augmentations and bug fixes

ID	Changes
1	Add around 20 unsupported instruction opcodes
2	Remove duplicated labels in generated assembly file [6]
3	Change input parsing logic to support output from newer `readelf`
4	Fix a bug in function alignments filtering
5	Add "-ocamlopt opt" to build script
6	Make some changes to support statically-linked binaries

TABLE III: Changes we made to Uroboros prototype

to perform a comparative evaluation on the Coreutils dataset and on CGC binaries. To the best of our knowledge, these changes and bug fixes, as listed in Table III, do not change the behavior and expected output of Uroboros. We will push these improvements upstream to the original Uroboros repository on GitHub.

Uroboros allows for the deactivation of some of its assumptions, which, as discussed previously, are overly restrictive for real-world cases. As we discussed previously in Section III-A, assumption 2 would prevent any modification of data sections. Therefore during evaluation, we enable assumptions 1 and 3 and disable assumption 2 (by specifying arguments `-a 3`) in order to obtain a comparative result.

For Uroboros, we use non-stripped binaries as input, as they rely on symbols in non-stripped binaries for function identification. `Ramblr` directly takes stripped binaries as input and carries out its own analyses to recover the necessary data.

D. Symbolization Correctness

First, we evaluate the *correctness* of `Ramblr`'s symbolization step on our dataset, with and without the use of its Fast Workarounds (implemented in `Ramblr Fast`). To do this, we collect the ground truth of mappings between labels and addresses from the linker `ld` during the *original* compilation of the binary, and compare this ground truth against the immediate values `Ramblr` symbolizes. It is important to note that `Ramblr` does not *utilize* the ground truth during its operation – it is only used for evaluation purposes. As we are interested in the potential damage to reassembled binaries, binaries that the tools *opted out* of reassembling were not included in this evaluation.

The mis-classifications represented by these results are roughly a measure of how *likely* the approach is to break the binary, as each mis-classification could result in a broken reference. When there are no mis-classifications in a given binary, the reassembled binary is guaranteed to work, except for in the scenarios described in Section VII-B. We could not make this evaluation comparative to Uroboros, as we were unable to extract this information from the Uroboros prototype.

The results are shown in Table IV. While both `Ramblr` and `Ramblr Fast` achieves an extremely low mis-classification rate, the former performs better than the latter, as expected.

E. Comparative Evaluation - Correctness

We compare `Ramblr` against Uroboros by evaluating both tools (plus `Ramblr` with Fast Workarounds) against our datasets. We run Uroboros, `Ramblr`, and `Ramblr Fast` on

[6]Clang displays error messages and terminates at duplicated labels, while GCC does not seem to care as long as duplicated labels do not reference different addresses.

Arch	Dataset	Opt. Level	Solution	Total References	False Negatives	False Negative %	False Positives	False Positive %
i386	CGC	O0	`Ramblr`	500682	0	0	0	0
			`Ramblr Fast`	500682	0	0	0	0
		O1	`Ramblr`	501613	0	0	0	0
			`Ramblr Fast`	501613	0	0	0	0
		O2	`Ramblr`	505409	0	0	0	0
			`Ramblr Fast`	505409	12112	2.39	15	0.02
		O3	`Ramblr`	505813	0	0	0	0
			`Ramblr Fast`	505813	12120	2.39	15	0.02
		Os	`Ramblr`	469512	3	0.0006	0	0
			`Ramblr Fast`	469512	12064	2.56	15	0.02
		Ofast	`Ramblr`	505828	0	0	0	0
			`Ramblr Fast`	505828	12120	2.39	15	0.02
i386	Coreutils	O0	`Ramblr`	128065	0	0	0	0
			`Ramblr Fast`	128065	0	0	0	0
		O1	`Ramblr`	124555	0	0	0	0
			`Ramblr Fast`	124555	0	0	0	0
		O2	`Ramblr`	122215	0	0	0	0
			`Ramblr Fast`	122215	0	0	0	0
		O3	`Ramblr`	192863	0	0	0	0
			`Ramblr Fast`	192863	4	0.0021	0	0
		Os	`Ramblr`	83600	1	0.0012	0	0
			`Ramblr Fast`	83600	1	0.0012	0	0
		Ofast	`Ramblr`	193317	0	0	0	0
			`Ramblr Fast`	193317	0	0	0	0
x86_64	Coreutils	O0	`Ramblr`	125005	4	0.00319	0	0
			`Ramblr Fast`	125005	4	0.00319	0	0
		O1	`Ramblr`	123156	4	0.0032	0	0
			`Ramblr Fast`	123156	4	0.0032	0	0
		O2	`Ramblr`	113651	4	0.0035	0	0
			`Ramblr Fast`	113651	4	0.0035	0	0
		O3	`Ramblr`	171302	4	0.0023	0	0
			`Ramblr Fast`	171302	4	0.0023	0	0
		Os	`Ramblr`	82592	4	0.0048	0	0
			`Ramblr Fast`	82592	4	0.0048	0	0
		Ofast	`Ramblr`	171849	8	0.0047	0	0
			`Ramblr Fast`	171849	12	0.0070	0	0

TABLE IV: Symbolization ground truth for different approaches across different datasets. Since reassembly failures are caused by mis-classification of symbols, we measure the rate at which symbols are mis-classified against ground truth provided by the linker during compilation.

each binary, and then run test cases against the reassembled binary to see if it still functions correctly. Using this data, we compile the rate of failures, which we present in Table V.

We evaluate both flavors of `Ramblr` on all optimization levels. However, Uroboros' failure rates increase to meaningless levels for optimizations above `O1`, so we only present `O0` and `O1` results. Furthermore, the Uroboros prototype that we initially used completely fails to reassemble 64-bit binaries[7]. Regardless of the reason, we were only able to carry out the comparative evaluation on 32-bit binaries. For CGC binaries, we evaluate all optimization levels on all tools.

As demonstrated in Table V, both `Ramblr` and `Ramblr Fast` are strictly better than Uroboros. With optimizations disabled, Uroboros breaks 22.64% of the Coreutils binaries, which is significantly worse than `Ramblr` and `Ramblr Fast`, which break *none*. Enabling optimization, this goes up to 56.61% for Uroboros and still none for `Ramblr`.

On the CGC dataset, Uroboros breaks 15% to 25% optimized binaries, which means that, when applied on real-world binaries, one out of four binaries will require manual inspection, intervention, and repair. For larger binaries, this is infeasible. By comparison, `Ramblr` achieves a success rate of over 98% across all levels, over 99% for optimization levels below `O3`, and 100% for unoptimized binaries.

[7]Upon contact, authors of Uroboros confirmed that there was a bug in their prototype, which causes misalignment of data sections in generated x86-64 assembly. Commit 45f018a was made to address this issue.

Correctness of Uroboros. The fact that Uroboros breaks many Coreutils binaries is unexpected, as it contradicts the claim in the Uroboros paper that no broken Coreutils binary was generated by Uroboros under *any* assumption. We investigated the issue, and found out the culprit was differing versions of GCC. Uroboros was evaluated on all Coreutils binaries compiled by the GCC version shipped with Ubuntu 12.04 LTS, which was GCC 4.6. Our Coreutils binaries are compiled by GCC 5.4.1. The prototype of Uroboros has trouble dealing with some memory references (e.g. `__JCR_LIST__`) in some binaries, and those references do not exist in ones generated by GCC 4.6. Additionally, GCC 5 introduces new optimizations that were not present in GCC 4, such as inter-procedural optimizations [9]. These optimizations more frequently produce hard-to-handle folded constants.

We reran the evaluation on Coreutils 8.15 compiled by GCC 4.6 shipped in Ubuntu 12.04 with the default optimization level (`O2`), and were able to reproduce their results. However, this reveals the fragility of the Uroboros approach. For instance, when running the evaluation on Coreutils 8.25 with `O0` under the same setting, we found out that `factor` was broken due to an incorrect symbolization in data sections.

Opt-out case study. `Ramblr` successfully detects the use of pointer encryption and decryption in `KPRCA_00044` and opts out, while `Ramblr Fast` fails to detect it, and generates a broken reassembled binary. `Ramblr` is the first binary reassembly engine with this detection capability. In fact, `Ramblr` was able to opt out of breaking all but one binary, resulting in a single broken binary out of the entire dataset.

Dataset	Opt. Level	Solution	Safety Opt-outs	Generation Failures	Test Failures	Successes	Total	Success %
CGC	O0	Uroboros	0	3	3	135	**141**	95.74%
		`Ramblr`	0	0	0	141		100%
		`Ramblr Fast`	0	0	0	141		100%
	O1	Uroboros	0	6	11	100	**117**	85.47%
		`Ramblr`	1	0	0	116		99.15%
		`Ramblr Fast`	0	0	1	116		99.15%
	O2	Uroboros	0	8	22	86	**116**	74.14%
		`Ramblr`	1	0	0	115		99.14%
		`Ramblr Fast`	0	0	2	114		98.28%
	O3	Uroboros	0	8	22	86	**116**	74.14%
		`Ramblr`	1	0	0	115		99.14%
		`Ramblr Fast`	0	0	2	114		98.28%
	Os	Uroboros	0	6	18	95	**119**	79.83%
		`Ramblr`	1	0	0	118		99.16%
		`Ramblr Fast`	0	0	2	117		98.32%
	Ofast	Uroboros	0	7	23	86	**116**	74.14%
		`Ramblr`	1	0	1	114		98.28%
		`Ramblr Fast`	0	0	2	114		98.28%
Coreutils	O0	Uroboros	0	0	24	82	**106**	77.36%
		`Ramblr`	0	0	0	106		100%
		`Ramblr Fast`	0	0	0	106		100%
	O1	Uroboros	0	5	55	46	**106**	43.39%
		`Ramblr`	0	0	0	106		100%
		`Ramblr Fast`	0	0	0	106		100%
	O2	`Ramblr`	0	0	0	106	**106**	100%
		`Ramblr Fast`	0	0	0	106		100%
	O3	`Ramblr`	0	0	0	106	**106**	100%
		`Ramblr Fast`	0	0	1	105		99.05%
	Os	`Ramblr`	0	0	0	106	**106**	100%
		`Ramblr Fast`	0	0	0	106		100%
	Ofast	`Ramblr`	0	0	0	106	**106**	100%
		`Ramblr Fast`	0	0	0	106		100%

TABLE V: The successes and failures of reassembling binaries, with six different optimization levels. The datasets used were the Coreutils binaries, compiled in 32-bit (due to limitations of the original Uroboros prototype), and the CGC binaries, which are all 32-bit. The column “Safety Opt-outs” represents the number of binaries for which the tool detected that functionality would be broken and opted out, “Generation Failures” refers to instances of the tool itself crashing during binary generation, and “Test Failures” conveys the number of reassembled binaries that failed functionality testing.

Binary	Opt. Level	Size	Code Size	Time	Time (Fast)
CROMU_00043	Os	93 KB	7.6 KB	35s	4s
NRFIN_00004	Os	344 KB	223 KB	37s	20s
EAGLE_00005	Os	5,408 KB	9.4 KB	75s	20s
NRFIN_00007	O3	233 KB	10 KB	73s	35s
KPRCA_00007	Os	91 KB	7.5 KB	93s	70s
NRFIN_00026	O0	10,768 KB	10,600 KB	525s	410s

TABLE VI: A comparison between the analysis runtimes of `Ramblr` and `Ramblr Fast` on some binaries in the CGC dataset.

F. `Ramblr` Runtime

`Ramblr Fast` trades functionality guarantees of the resulting binaries for improved speed of the reassembling process. For most binaries in our dataset, this is irrelevant. In fact, the median runtime of `Ramblr Fast` in the CGC dataset is 2.8 seconds, compared to 3.0 seconds for `Ramblr`. However, `Ramblr Fast` scales considerably better for large binaries.

In Table VI, we discuss the relative runtime of `Ramblr` versus `Ramblr Fast` for the biggest CGC binaries in our dataset. Our fast workarounds significantly decrease runtime in all cases, but it is important to note that runtime is not completely contingent on binary size, but rather, on the amount of code that must be analyzed by the data classification and symbolization steps of the systematic `Ramblr` approach.

G. Execution Overhead and Binary Size

In Section 6.2.1 of the Uroboros paper, the authors report that binaries produced by Uroboros have execution time overheads of up to 7 percent (although the average was under a percent). We evaluated the binaries produced by `Ramblr`, but identified *no* execution overhead. For most purposes, the binaries are perfect replacements for the originals.

Likewise, Uroboros introduced a small increase in size for their Coreutils dataset. Since unimportant sections (like `.comment`) are removed by `Ramblr` during reassembling, our binaries are usually smaller than the originals. Compensating for this removal, the resulting size is practically identical.

XI. Discussion

While our approach improves the feasibility of real-world binary reassembly, it is a long way from “solving” the general issue. To focus the community’s attention on potential future work in this field, we detail what we feel are the biggest limitations of our technique in this section.

The infeasibility of static content classification. We admit, and would like to stress again, that as Horspool, et al. maintained, static content classification is infeasible [11]. Our reassembling approach is an empirical solution that works on many binaries whose integer distributions roughly follow the pattern as presented in Section VII. Obviously, an easy way for anti-reassembling is basing the binary to another base address during linking, so immediate values belonging to memory region of binaries collide with normal immediate values in the binary. In that case, our approach will most likely fail and result in broken binaries.

CFG recovery. The performance of CFG recovery may work differently on binaries holding different features. The technique on which our CFG recovery is based works well

on our tested Linux binaries compiled with GCC or Clang, which do not generate any inline data [1]. Some compilers (like MSVC) puts inline data into executable regions of binaries, most notably, jump tables. While we believe our CFG recovery and disassembly technique will work on such binaries with the help of content classification, more work is needed in that direction.

XII. Conclusion

We presented `Ramblr`, a tool for the disassembly, modification, and reassembly of binaries. The proposed approach extends previous approaches to the problem of reassembling binaries, making it possible to apply static binary modifications to real-world binaries, even when compiler optimizations are used. `Ramblr` uses a novel composition of static analyses to characterize data contained in a binary, allowing for an improved symbolization. In addition, the reassembly process introduces no execution overhead in the resulting binary. The ability to modify binaries without affecting their performance opens a number of applications, ranging from efficient instrumentation to binary hardening.

Acknowledgements

We would like to thank all contributors to the DARPA Cyber Grand Challenge organization (for providing an excellent testing dataset for `Ramblr`), the contributors of angr, and all our fellow Shellphish CGC team members. This material is based on research sponsored by the Office of Naval Research under grant number N00014-15-1-2948 and by DARPA under agreement number N66001-13-2-4039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work is also sponsored by a gift from Google's Anti-Abuse group.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries," in *25th USENIX Security Symposium (USENIX Security '16)*. Austin, TX: USENIX Association, 2016, pp. 583–600.

[2] G. Balakrishnan and T. Reps, "Analyzing Memory Accesses in x86 Executables," in *International Conference on Compiler Construction*, 2004, pp. 5–23.

[3] A. R. Bernat and B. P. Miller, "Anywhere, Any-Time Binary Instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE '11)*, 2011, pp. 9–16.

[4] D. L. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *International Conference on Computer Aided Verification*, vol. 6806 LNCS. Springer, 2011, pp. 463–469.

[6] Cryptic Apps, "Hopper," https://www.hopperapp.com/.

[7] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen, "On the Static Analysis of Indirect Control Flow Transfers in Binaries," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Application*, 2000, pp. 1013–1019.

[8] M. Egele, M. Woo, and D. Brumley, "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components," in *23rd USENIX Security Symposium (USENIX Security '14)*. San Diego, CA: USENIX Association, 2014, pp. 303–317.

[9] GCC, "GCC 5 Release Notes," https://gcc.gnu.org/gcc-5/changes.html.

[10] L. C. Harris and B. P. Miller, "Practical Analysis of Stripped Binary Code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.

[11] R. N. Horspool and N. Marovac, "An Approach to the Problem of Detranslation of Computer Programs," *Computer Journal*, vol. 23, no. 3, pp. 223–229, 1980.

[12] J. Kinder, "Static Analysis of x86 Executables," Ph.D. dissertation, 2010.

[13] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, 2007, p. 89.

[14] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting Security in COTS Software with Binary Rewriting," *IFIP Advances in Information and Communication Technology*, vol. 354, pp. 154–172, 2011.

[15] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, "N-version Disassembly: Differential Testing of x86 Disassemblers," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*, 2010, p. 265.

[16] pancake, "radare," http://www.radare.org/r/.

[17] N. A. Quynh. (2016) The Ultimate Disassembly Framework Capstone. [Online]. Available: http://capstone-engine.org

[18] T. Reps and G. Balakrishnan, "Improved Memory-Access Analysis for x86 Executables," *International Conference on Compiler Construction*, vol. 4959 LNCS, no. i, pp. 16–35, 2008.

[19] s3team. (2015) s3team/uroboros: Infrastructure for Reassembleable Disassembling and Transformation (v 0.1). [Online]. Available: https://github.com/s3team/uroboros

[20] Shellphish, "DARPA CGC," http://shellphish.net/cgc/.

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "(State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the IEEE Security and Privacy*, 2016, pp. 138–157.

[22] M. Smithson, K. Elwazeer, K. Anand, A. Kotha, and R. Barua, "Static Binary Rewriting without Supplemental Information: Overcoming the Tradeoff between Coverage and Correctness," in *Proceedings - 20th Working Conference on Reverse Engineering (WCRE 2013)*, R. Lämmel, R. Oliveto, and R. Robbes, Eds. Koblenz, Germany: IEEE, 2013, pp. 52–61.

[23] Vector 35, "binary.ninja : a reversing engineering platform," https://binary.ninja/.

[24] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries," in *Proceedings of 2015 Annual Computer Security Applications Conference (ACSAC '15)*, 2015, pp. 331–340.

[25] S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," in *24th USENIX Security Symposium (USENIX Security '15)*. USENIX Association, 2015, pp. 627–642.

[26] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing Untrusted Code via Compiler-Agnostic Binary Rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, 2012, p. 299.

[27] R. Wartell, V. Mohan, K. W. Hamlen, Z. Lin, and W. C. Rd, "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, 2012, pp. 157–168.

[28] J. Zeng, Y. Fu, K. a. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation Resilient Binary Code Reuse through Trace-oriented Programming," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, 2013, pp. 487–498.

[29] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22nd USENIX Conference on Security (USENIX Security '13)*, 2013, pp. 337–352.