GRIN: Make Rewriting More Precise

Linan, Tian
Institute of Information Engineering, Chinese Academy of Sciences, School of Cyber Security, University of Chinese Academy of Sciences Beijing, China
tianlinan@iie.ac.cn

Liwei, Chen*
Institute of Information Engineering, Chinese Academy of Sciences, School of Cyber Security, University of Chinese Academy of Sciences Beijing, China
chenliwei@iie.ac.cn

Delin, Kong
Ruixin Academy of Classic Learning, Beijing Institute of Technology, Beijing, China
1120210705@bit.edu.cn

Gang, Shi
Institute of Information Engineering, Chinese Academy of Sciences, School of Cyber Security, University of Chinese Academy of Sciences Beijing, China
shigang@iie.ac.cn

ABSTRACT

In computer security, many systems and applications depend on binary rewriting techniques when source code is absent, including binary instrumentation, profiling and security policy reinforcement. While the rewriting technique is continuously evolving, many static binary rewriters are still unable to correctly disassemble and accurately cover all legal instructions. Dynamic binary rewriters can achieve accuracy, but are not able to guarantee a full-coverage. Therefore, existing binary rewriting techniques do not meet all the requirements for binary rewriting, and make various assumptions for their application purposes. In this paper, we present GRIN, a novel and practical binary rewriting tool that can accurately identify each legal instruction, while guaranteeing a code full-coverage. We design a dynamic execution technique, which can identify each legal instruction. Also, we develop a branch backtracking technique, which can address various challenges of identifying Our tool does not require any assumptions and relocation information, and can be applied to all security applications. We have implemented a prototype of GRIN and evaluated the SPEC2006 and the whole set of GNU Coreutils. The experiment results show that the average instruction redundancy for SPEC is only 0.135% and for Coreutils is 0.062%.

CCS CONCEPTS

• **Security and privacy**; • **Software and application**; • **Software reverse engineering**;

KEYWORDS

Binary Rewriting, Static Analysis, Dynamic Execution

*Corresponding author.

ACM Reference Format:
Linan, Tian, Liwei, Chen, Delin, Kong, and Gang, Shi. 2022. GRIN: Make Rewriting More Precise. In *2022 3rd Asia Service Sciences and Software Engineering Conference (ASSE) (ASSE' 22), February 24–26, 2022, Macau, Macao.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3523181.3523207

1 INTRODUCTION

Binary rewriting is a core technology in computer security, which can transform binary without source code. It has many application scenarios, including enforcing security policy, binary code hardening and recovering the call graph (CG) and function boundary. A common key point of these application scenarios is the need to accurately identify the memory addresses of all legal instructions when rewriting binaries. But accurate rewriting identification technique is still an unsolved problem.

Identifying memory addresses of all instructions is a significant indicator of the sophistication of rewriting technology [3], and has various categories: (1)To recognize code and data [9]. Due to hand-written code or compiler performance optimizations, they mixed in-line data within code section. It is very difficult to distinguish them in CISC architecture (e.g., x86); (2)To identify static memory addresses. Since existing alignment-related padding [46], [39] in the variable-length instruction set, and there is no distinction between instructions and data addresses in executable, the correctness of rewriting cannot be guaranteed if it is not correctly identified; (3)To recognize dynamic memory addresses and data, containing indirect control-flow addresses whose addresses need to be computed dynamically through the form of base address plus index and offset (e.g., jump table, x86 assembly "jmp [eax]"), code pointers(they will be assigned a valid address at runtime, e.g., function pointer), and dynamically determined addresses data(e.g., function pointer arguments in callbacks [5]).

To address these challenges, current binary rewriting techniques had made significant efforts, but still do not fulfill requirements. BIRD [24] can disassemble all legal instructions in a binary file, but its approaches are only used for binary hardening and profiling. BINCFI [46] is the first practical rewriting technique to handle memory addresses for binary security. BINCFI presents an approach that combines linear and recursive disassembling to recognize instruction addresses. However, it cannot handle mixed code and data and still need a copy of code for rewritten binary. SecondWrite [2] and Zipr [18] are both lift the binary to the intermediate representation

(IR) when the transformation phase of rewriting. They leverage multiple static analysis techniques to disassemble binaries, but they are not guaranteed to accurately cover all legal instructions in the binary. UROBOROS [39] recognizes memory addresses by a set of heuristics and can relocate instructions when rewriting binary, but it still cannot be precise to identify code pointers. REV.NG [16] lifts binary into LLVM IR [21] by using QEMU tiny code generator (TCG) [6] lifter, and which retrieves these IRs and recognizes possible memory addresses, realizing a multi-architecture rewriting system. But it still lacks precision in determining indirect memory addresses. Recently approach MULTIVERSE [5], that uses an iterativeoffset disassembly from binary start address combined with lookup tables to solve the problem of recognizing addresses. However, it has a high memory overhead and also brings unnecessary compatibility issues (rewriting the libraries).

According to our in-depth investigation, many existing tools can recover data and control structures from executables, but considering the perennial problem of the distinction between data and code, or handling dynamically computed address and data, the state-of-the-art rewriting tools still require many assumptions and incur high overhead. In particular, when rewriting the stripped binaries without symbol and relocation information, existing techniques still cannot accurately recover all legal instructions.

In this paper, we propose a novel, practical binary rewriting tool called GRIN. In GRIN, we develop two key techniques to address the challenges of accurately rewriting. We propose a dynamic execution technique, which executes each path of the binary program and identifies memory addresses of the executed instruction. In addition, we also can recognize dynamically computed addresses and data by the execution state information of each execution path. Meanwhile, unlike the previous dynamic execution techniques of providing random input when program inputs are required, we leverage the execution state information in combination with practical inputs to ensure program normal execution in the explored path and avoid program crashes as possible.

Based on the instruction memory address we identified, we lift the legal instruction to an IR and then we transform IR to the rewritten executable. Moreover, to cope with program crashes during path exploration, we propose a branch backtracking technique, which can determine an actual execution path where the crash is located. Along this path, we perform use-def analysis and data flow analysis based on lifted IRs, which can quickly and easily supply a valid value to program crash points and guarantee code path coverage. Therefore, our approach can easily and completely identify memory addresses of instructions.

In summary, our main contributions are as follows:

- We present GRIN, a novel dynamic binary rewriting system that can precisely identify the address of any instruction in an executable and rewrite binary both accurate and complete.
- We design a dynamic execution technique that does not require any assumptions and relocation information to identify address but instead executes each path of an executable to obtain the precise address of the legal instruction, meanwhile, guaranteeing code coverage.
- We develop a branch backtracking technique, which can determine an actual execution path. Based on this sensitive execution path, we use the use-def analysis for backward retrieving to recover program crashes and handle indirect addresses.
- Our approach does not consider relocation issues and can be applied to all security applications. We have implemented a prototype of GRIN and evaluated it with the SPECint CPU 2006 benchmark suite and all of the GNU Core Utilities binaries. The experiment results show that GRIN can correctly rewrite all binaries we have tested and substantially improve the accuracy of rewriting.

2 RELATED WORK

The binary rewriting work began in the 1960s. One of the first static rewrites is Honeywell [20], which is used to translate programs from the IBM 1400 series machines to Honeywell's Series 200 system for emulation. The earliest dynamic binary rewriting system [7] is implemented in 1987, which construct a special node mapping table to retarget indirect branch targets. Afterwards, there are many binary rewriting systems that are implemented for code and performance optimization (e.g., Hollingsworth et al. [19], ETCH [32]), instrumentation and observation for profiling (e.g., ATOM [36], Valgrind [25]), emulation [6], and architecture translation (e.g., TIBBIT [10], Sites et al. [35]).

However, most of the early rewriting work rewrites binaries are RISC architectures, which instructions are a fixed length ISA and have a well-aligned. But for CISC architecture, which is a variable-length instruction set (e.g., x86), identifying and rewriting them may not be easy and is more challenging. Although ETCH [32] operates target is binaries of x86 architecture, it is mainly used for performance optimization and does not rewrite all binary instructions instead by static minimal-invasive alteration to instrument code, which cannot solve the problem of identifying and rewriting instructions. As such, early approaches do not meet the requirements of rewrites.

BINCFI [46] is the first practical rewriting system that can handle memory addresses and enforce control-flow integrity into binaries. BIINCFI presents a combination of linear and recursive disassembling approaches to recognize instruction memory address, but it still fails when handling mixed code and data. UROBOROS [39] summarizes a set of heuristics to identify binary memory addresses. It implements the reassembleable disassembling process by using the same disassembling algorithm from BINCFI [46]. UROBOROS assumes all static ICF targets stored in data sections are n-byte aligned and point to the function entry or jump table entries. However, it has false positives and false negatives for its approach to datatype identification.

SecondWrite [2] lifts the disassembled code of binaries into LLVM IR, then rewriting and converting back the executables. SecondWrite can solve the problem of identifying ICF target addresses by run-time checking metadata tables that store the mapping for the old ICF target and its corresponding new address [27]. SecondWrite identifies the static memory addresses by leveraging multiple static analysis techniques, but cannot still accurately identify each instruction. Zipr [18] leverages existing multiple disassemblers to

```
1  int (*foo)(int, int (*)(int, int));
2  int (*op[2])(int, int) = {lsh,rsh};
3
4  void test(int select, int p)
5  {
6      if(select==1)
7        foo = mode1;
8      else if(select==2)
9        foo = mode2;
10
11     result = foo(p, op[p]);
12 }
13
14 int lsh(int a, int b){
15     return a<<b;
16 }
17 int rsh(int a, int b){
18     return a>>b;
19 }
```

(a) Source code of example program

```
1  mode1:
2   pushq %rbp
3   movq %rsp, %rbp
4   pushq %rbx
5   subq $40, %rsp
6   movl %edi, -36(%rbp)
7   movq %rsi, -48(%rbp)
8   ...
9   jmp _b1
10 _db:
11  .long 10
12 _b1:
13  addl _db, %eax
14  ...
15  movq -48(%rbp), %rax
16  movl %ecx, %esi
17  movl %edx, %edi
18  call *%rax
19  ...
```

(b) Source code of mode

```
40027e <test>:
40027e: push   %rbp
...
40028c: cmpl   $0x1,-0x4(%rbp)
400290: jne    40029f <test+0x21>
400292: movq   $0x400307,0x20622b(%rip)
40029d: jmp    4002b0 <test+0x32>
40029f: cmpl   $0x2,-0x4(%rbp)
4002a3: jne    4002b0 <test+0x32>
4002a5: movq   $0x40035b,0x206218(%rip)
4002b0: mov    0x206211(%rip),%rax
4002b7: mov    -0x8(%rbp),%edx
4002ba: movslq %edx,%rdx
4002bd: mov    0x605610(,%rdx,8),%rcx
4002c5: mov    -0x8(%rbp),%edx
4002c8: mov    %rcx,%rsi
4002cb: mov    %edx,%edi
4002cd: callq  *%rax
...
```

(c) Binary code of test

Figure 1: An artificial example that illustrates the identifying challenge in rewriting.

solve the problem of identifying static memory addresses. In recent work MULTIVERSE [5], we have discussed their approach and compared it with GRIN in section V.

Compared with the existing works, GRIN can provide a practical solution to precisely identify each static memory address of an instruction, thereby ensuring the correctness of rewritten executables. It also does not require relocation information, can be used in binary code hardening, instrumentation, profiling, security policy enforcement.

3 CHALLENGES AND OVERVIEW

As mentioned in the introduction, we have discussed the technical challenges of identifying the memory addresses of all legal instructions in rewriting. In this section, we introduce an example to further illustrate these challenges clearly, moreover provide an overview of our rewriting system.

3.1 Rewriting Challenges

As an artificial example of the challenge of accurately identifying all legal instructions in rewriting, consider the snippet program as shown in Figure 1. It shows that function pointer **foo** is assigned by function **mode1** or **mode2**, in which **foo** also pass a function pointer argument that is stored in an array of pointers **op**. Therefore, when rewriting our example binary file, we will encounter two dynamically computed indirect control-flow (ICF) target addresses at line 18 of Figure 1(b) and at 0x4002cd of Figure 1(c). In addition, the function mode(1/2) received by **foo** is implemented in assembly, which shows an example of in-line data within instructions in lines 9-13 of Figure 1(b).

In our example, the ICF instruction **call** is assigned within intra-procedural (as shown in Figure 1(c)), which receives function addresses (mode 1 or 2) through two **mov** instructions at address 0x400292 or 0x4002a5. In fact, the target address of this ICF instruction is determined by the stack variable **select**, which is assigned a specific value at runtime (as shown in Figure 1(a)). Another ICF instruction is assigned within inter-procedural (as shown at line 18 of the source code for **mode**), which receives function pointer argument (**rsi** at line 7), which has a value of **lsh** or **rsh**, and is specified by stack variable **p.** As shown in the binary code of **test**, **p** is an index value of a jump table (at 0x4002bd), which is also determined at runtime (**rdx** at 0x4002b7 is set by stack, and assign the value of the array of **op** of the **rdx** index at 0x4002bd to **rcx**).

In existing static rewriting tools, it is difficult to identify the ICF target address of our example, because these values need to be determined dynamically. For the ICF instruction assigned in intra-procedural (in Figure 1(c)), static predicting analysis can retrieve the ICF target addresses and ultimately analyzes them as being derived from the assignment operation at address 0x400292 or 0x4002a5. However, considering the situation of function **mode** in Figure 1(b), static analysis techniques have difficulty identifying the targets of the ICF instruction, because statically analysis is difficult to determine they are computed at 0x4002bd in function **test**. Especially, for stripped binary files without symbol and relocation information, retrieving and identifying the address of ICF target instruction will become intractable. Moreover, in function **mode**, static data is mixed within instructions in lines 9-13. When identifying memory addresses of instructions, linear disassembly would misinterpret these encoding and yield the following garbage instructions.

```
40032a:  jmp    400330
40032c:  or     (%rax),%al
400330:  add    0x40032c,%eax
```

Figure 2: Garbage instructions.

Such a potential gap can be bypassed by recursive disassembly, which jumps from line 9 to line 13 along the control flow to avoid identifying data in the code. However, when it executes to line 18, it will be difficult to determine the value of ICF targets (the value of **rax**). In a more complicated situation, if the jump instruction at line 9 is also an ICF transfer instruction, recursive disassembly

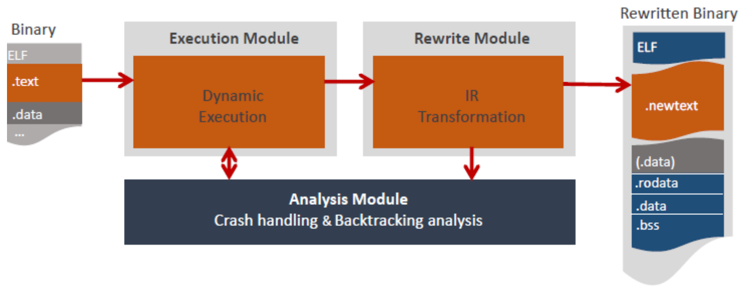

Figure 3: System Overview.

will be difficult to skip the data mixed in the code and identify legal instructions.

In GRIN, we make binary "alive". We first make our example program execute normally from the entry address. If a branch is encountered (**if** and **else if** in source code of **test** and **jne** in binary code of **test**), GRIN records the not taken branch address and current memory state, and continues execution. When the first execution ends, GRIN will restore the recorded memory state and execute not taken branch. Assume that **if** branch is executed in the first run, we record the **else if** branch and explore this path later. Eventually, GRIN identifies that the ICF target address at 0x4002cd has two values (0x400307 and 0x40035b). While the approximate technique is proposed by FXE [42] to construct precise control flow graphs from binaries, it does not attempt to rewrite binary and make it runnable. In GRIN, we do not need to consider complicated algorithms (e.g., handling loop), we only care about the correctness and completeness of rewritten binaries.

When the **if** branch is taken, we explore **mode1** function and assume that the variable **p** is zero. When in-line data within instructions are encountered, GRIN will follow the control flow, jumping from line 9 to line 13 to identify legal instructions. When GRIN executes to line 18, the **rax** is assigned the address of **lsh** function. In the path exploration just performed, GRIN lifts each executed instruction to IR. Thus, GRIN analyzes the IR generated by executing this path and performs a backward retrieve. Finally, GRIN retrieves another value of **rax** at address 0x4002bd, which is the address of the **rsh** function. We record this address(including current CPU and memory states) and explore it later.

3.2 System Overview

As described above, the system architecture of GRIN is shown in Figure 3. Our rewriting system follows the following design principles:

- Execute every piece of code in the binary.
- Make full use of all available data.
- When crashed, assign a valid value.

We run every piece of code in the binary (in execution module) and lift each executed instruction to IR. In order to keep running without crashing, we make full use of the state data generated when exploring this path. Once it crashes, we analyze the IR just generated on this path and get a valid value to patch crash (in analysis module). In addition, we analyze IRs and combine current execution information to obtain more unexplored branch addresses. When all paths are explored, we will transform all IRs to the **newtext** section of rewritten binary (in rewrite module). The original data and code section will be copied and as part of **data** section of this binary data section. Finally, we will create the new rewritten binary.

Algorithm 1 Dynamic Execution

```
input : entry point address of binary entry
input : empty executed blocks set instructions
input : empty unexplored target addresses container branches
output: executed code basic blocks are in instructions
definition: program counter pointer pc, the have executed address set addresses;
1 pc ← entry;
2 while execution flow ≠ exit do
3    blocks ← translate(pc);
4    if pc ∉ addresses then
5       instructions ← instructions ∪ blocks;
6       branches ← RecordBranch(blocks);
7       addresses ← addresses ∪ pc;
8    pc ← update(pc);
9 while branches do
10   pc ← branches.pop();
11   while vaild(pc) do
12      if pc ∈ addresses then
13         break;
14      blocks ← translate(pc);
15      if crash then
16         handleCrash();
17         break;
18      instructions ← instructions ∪ blocks;
19      branches ← RecordBranch(blocks);
20      addresses ← addresses ∪ pc;
21      pc ← update(pc);
```

4 IDENTIFYING AND REWRITING

4.1 Dynamic Execution

In order to identify the memory address of each instruction, we dynamically execute every instruction in the binaries. However, it will incur poor covered paths drawback. Here, our dynamic execution technique can explore different paths of a given binary. While guaranteeing code coverage, this approach solves the various challenges to identify instruction addresses. During execution, it will

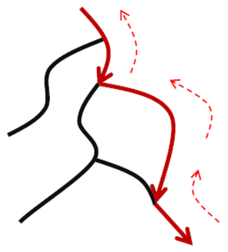

Figure 4: An actual execution path.

inevitably explore repeated paths and identify many duplicate instructions. As our goal is to rewrite binary instructions, so we check the address of the next code block of instructions to be executed. If the next address has been executed, we will stop executing this path and explore the next path. This process work is shown in algorithm 1

We make the binary program run normally from the entry point (line 1). We execute the program with basic block-grained (line 3), these blocks end with a control transfer instruction. If the block we retrieved contains a conditional transfer instruction (e.g., **jcc** in x86), we will record the address of the not taken branch and store current CPU and memory states (line 6). In lines 4-8, we save the executed code block in **instructions**, and according to the current block execution result, we update the **pc** to the address of the next code block to be executed. Then, we check whether the **pc** has been executed (condition $pc \notin addresses$ at line 4), if so, we will keep the code blocks run to continue, and do not save this block and any state.

In the first exploration, once the execution is over, we will capture the exit signal and explore the branch storing in **branches** (line 2 and 9). We update the **pc** to the unexplored path address (line 10) and continue to execute from this address. In lines 14-21, if the executing code block contains branch instruction, we still record the address of not taken branch and the current runtime environment (at line 19). When a code block is executed, we will mark the end instruction type of this block in **translate** (e.g., indirect jmp and call in x86), which can provide convenience for handling the jump table later. When an error occurs in the execution path and the program crashes, we analyze the cause of the error and further locate the crash point. We analyze the source of the crash value and assign a valid value to it (detailed in Crash Analysis). Then we break out of the loop and continue our exploration from the crash location (in lines 15-17, and update **pc** at line 10). Note that each time we update the **pc**, we will check whether the address has been explored in lines 12-13. Once this address has been done, we terminate exploring and continue to update the **pc**. As soon as all the branch addresses in **addresses** are executed, we traverse all the paths of the binary. Because in software design, most program paths are inter-related and reachable. Therefore, by traversing all the branches recorded when exploring each path, we can guarantee complete path coverage.

4.2 Backtracking Analysis

In order for the path exploration to execute normally when the program crashes or needs to identify other branch ad dresses, we perform a backtracking analysis along the path just executed. Since we explore the binary paths in a depth-first fashion, when we retrieve a branch path, we will retrieve all the branch paths related to this branch. In other words, our branch backtracking technique will find a sensitive path in the binary control-flow graph, as shown in Figure 4. This sensitive path is an actual execution path from the program entry point during dynamic execution (due to the repeated address, some paths will stop exploring, so the repeated part is not counted as the sensitive path of this path). Based on it, we perform a usedef analysis, which can easy to patch crash points or identify more indirect addresses. To this end, tracing sensitive paths allows us to always trace one valid execution path, preventing path explosion.

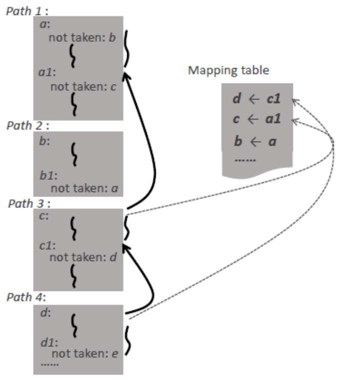

Figure 5: A lookup example for Branch Backtracking technique.

In order to get the actual execution path of one branch. We use a mapping table to record the address relationship between the starting address of a branch path and the location of the conditional branch that yielded this not taken address. When a new path is executed, we simply map the start address as an lvalue of an item and map the conditional branch address as rvalue of this item. In addition, if this branch has been explored, we also will not record the repeated mapping relationship, which is the same as the dynamic execution algorithm.

Figure 5 shows a backtracking analysis example for an execution path. When executing path one, two unexplored branches will be yielded. So we map the address relationship of branch **b** and execute it later as path two. Then the same operation to branch **c** and **d**, we explore them as path three and four. So, when we perform a backtracking analysis starting from path four, we retrieve to the entry address of the current path along path four. We then look up the mapping table, and through mapping relationship we find that the instruction address **c1** points to the branch starting address **d**. Then we switch the backward path and continue to retrieve to the starting address **c** from the address **c1** of path three. We look up the mapping relationship in our mapping table and continue to retrieve basic block from the address **a1** of path one. As such, we can obtain the sensitive execution path of path four. Therefore, by performing a lookup in the mapping table, we can flexibly get an actual execution path of any branch.

```
mov rdi,(rax)  %2 = load i64, i64* @rdi
               %3 = load i64, i64* @rax
               %4 = inttoptr i64 %3 to i64*
               store i64 %2, i64* %4
```

Figure 6: IR example.

4.3 Crash Analysis

When exploring each branch path, the illegal memory state may cause the program to crash and terminate the current path exploration. In order to restore a valid memory state as much as possible, we leverage backtracking analysis technique to trace the source of the crash value.

Since our IR lifting process is based on instruction semantics. Therefore, by analyzing the IRs of the crashed code block, we can determine the program crash point, such as illegal access to the memory address. The IR for an access memory instruction lift is as follows.

Operand **rax** has an operation to access memory, and its corresponding IR forms are **load** and **inttoptr** instructions. Therefore, once the explored path crashes, by retrieving lifted IRs, we match the load-inttoptr dependency. When determining this IR dependency, we check if it is legal access memory address based on its current memory execution states. If this address value is illegal, we will backtrack along the actual execution path of this branch path to analyze the source of this value. We divide the source of illegal value into two categories: (1)illegal value derives from register assignment; (2)the value derives from memory address.

For the value derived from register assignment, we perform a backward retrieve. When we encounter the assignment of stack register (e.g., **rsp**), we stop the backtracking analysis and identify that this illegal value is a local variable of this executable. As such, we pick the current stack address at random as the legal value and assign it to the crash point. Then continue exploring from the crash location of the path just explored.

If the illegal value we retrieve derives from the assignment of memory address, we will identify that this illegal value derives from a global variable. So unlike the above assignment, we assign a legal value to the crash point as well as the correlated memory location. Specially, if the memory location retrieved by backtracking analysis is a read-only memory, we will not assign random legal value. If this memory belongs to the heap, we will allocate some more heap memory while assigning legal value, preventing illegal access due to unallocated heap memory. By assigning a legal value to correlated memory locations, we can reduce the frequency of program crashes when other unexplored instructions operate on this memory, guaranteeing path execution.

5 IMPLEMENTATION

We have implemented above approaches in a system called GRIN, which was implemented and evaluated on Ubuntu 18.04 x86 64 GNU/Linux. We rewrite target is 64-bit x86 binaries running atop the Linux system. Currently, we focus on statically linked executables and the reason for this will be discussed below. The implementation of GRIN is based on REV.NG [16], which is a static binary translator. It employs the QEMU TCG translates binaries to LLVM IR. Therefore, based on their translation work, we develop our dynamic execution and branch backtracking techniques.

Based on the instructions identified by exploring each path, we lift each executed block of code to IR. We map all rewritten code block addresses into the mapping table. When we transform each code block, if the end of the code block with a direct control flow transfer instruction, we insert a branch instruction at the end of the code block, which allows the block to jump directly to the next code block at runtime. If the code block does not end with a direct transfer instruction, we will jump directly to the mapping table and look up the target address corresponding to the actual address at runtime.

6 EVALUATION

We evaluate GRIN for correctness and cost. We first evaluate whether GRIN can accurately identify the instructions in a binary, and compare it to other tools. Then we report the cost of GRIN, which reveals the size expansion of the rewritten binary. We use SPEC CPU 2006 benchmark suite, GNU Core Utilities binaries, and real-world binaries to evaluate GRIN.

6.1 Correctness

To evaluate the correctness of our rewriting system instruction identification, we verified from the following two aspects. First, we summarize some representative instructions of rewritten binaries and compare them against MULTIVERSE. Second, We run the rewritten executables and test their functionality.

We verify the correctness by comparing the number of control-transfer instructions (**jmp, call** and **ret**) that GRIN identifies during rewriting. However, that was already a challenge to identify all control-transfer instructions. Here, we use Objdump to disassemble the unstripped binaries and count the number of control-transfer instructions. Then, GRIN rewrites the stripped binaries and records the control-transfer instruction it identifies. Table I shows the result of using GRIN to identify SPEC binaries. By comparison, GRIN's redundancy rate does not exceed 0.4%. The average redundancy for SPEC is 0.135%. As expected, the identification results of GRIN and Objdump are approximately equal. However, the redundancy for **471.omnetpp** looks relatively high. We suspect that inline data within instructions incur overhead because we do not mark them, but identify them as part of the code. Other examples with redundancy have the same reason.

We also list the results recorded by MULTIVERSE in Table I. We compare the results of MULTIVERSE rewriting dynamically linked binaries. As shown, although MULTIVERSE guarantees the correctness of the rewriting, the implementation of its design will identify a large number of redundant instructions, and theoretically, its identification results should be less.

To verify correctness, we also tested the functionality of rewritten executables. We executed both the rewritten version and the original version of binary with default input configuration. We tested 12 SPEC2006 benchmark programs using the test input set they provided, and 105 GNU Coreutils using all the command arguments they specified. We also tested 5 realworld binaries including a web server, compressing tool and calculator, etc (bc, ctags, gzip, nweb and thttpd). We compared the output of both Coreutils and

Table 1: Results of using GRIN to identify SPEC binaries, and compared with MULTIVERSE and Objdump (referring to identification results in Objdump)

Benchmark	Call			Jump			Ret			Redundancy
	Objdump	GRIN	MULT.	Objdump	GRIN	MULT.	Objdump	GRIN	MULT.	
400.perlbench	15700	15700	34784	13232	13236	29220	2130	2130	22306	0.01%
401.bzip2	514	514	1270	603	603	1202	177	177	874	0.00%
403.gcc	46439	46438	119038	25060	25084	80212	6281	6281	45410	0.03%
429.mcf	387	387	323	493	493	294	161	161	250	0.00%
445.gobmk	9377	9377	27098	4842	4842	18426	3221	3221	20918	0.00%
456.hmmer	4020	4020	8576	2270	2271	5608	829	829	4106	0.01%
458.sjeng	1371	1371	2822	1441	1441	2996	341	341	1570	0.00%
462.libquantum	685	685	1188	492	492	904	219	219	812	0.00%
464.h264ref	3451	3451	8906	3084	3088	8518	800	800	6318	0.06%
471.omnetpp	24572	24528	37408	10037	10212	11814	3974	3979	14326	0.35%
473.astar	1982	1982	1074	1717	1725	712	469	469	750	0.19%
483.xalancbmk	76301	76301	154546	31808	32992	73308	14091	14091	75674	0.97%

Table 2: Statistics of SPEC binaries identified by GRIN

Benchmark	Ind. Jumps	Dir. Jumps	Ind. Calls	Dir. Calls	Cond.Branches	Orig.Size (kB)	New Size(kB)	Inc.
400.perlbench	101	13135	224	15476	30000	2217	48579	21.91x
401.bzip2	9	594	32	482	1799	397	3355	8.45x
403.gcc	496	24588	802	45637	77158	4200	98384	23.43x
429.mcf	9	484	17	370	1349	444	2469	5.56x
445.gobmk	22	4820	114	9263	15843	4562	33260	7.29x
456.hmmer	35	2236	41	3979	7491	940	11899	12.66x
458.sjeng	24	1417	18	1353	3912	616	8217	13.34x
462.libquantum	8	484	19	666	1345	562	3103	5.52x
464.h264ref	26	3062	571	2880	9940	1134	19041	16.79x
471.omnetpp	449	9763	3928	20600	18645	6647	40664	6.12x
473.astar	46	1679	210	1772	4391	1253	6858	5.47x
483.xalancbmk	2305	30687	27045	49221	76266	11842	173241	14.63x

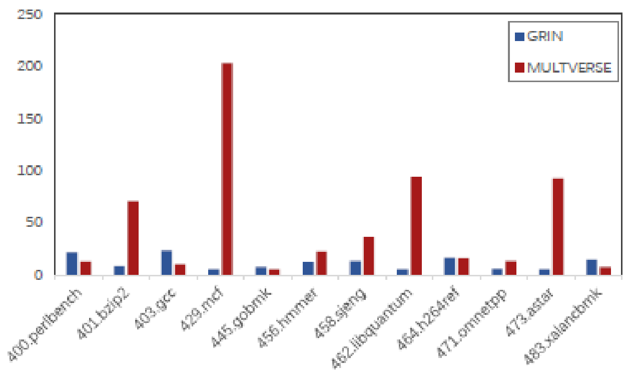

Figure 7: A comparison of GRIN and MULTVERSE code expansion.

SPEC with original binaries. In all of our tests, their execution behavior and output result are as expected.

6.2 Cost

In order to evaluate the cost of GRIN, we summarize the statistics of rewritten binaries. Table III shows the statistics of SPEC binaries and also shows how the GRIN rewriting system affects the file size of each benchmark binary. The average size expansion for SPEC is 11.76 times. We analyzed the main reasons for the increase in size and have two speculations: the additional helper library in rewritten binary and IR transformation when rewriting.

Our current implementation of the dynamic execution technique is based on QEMU's TCG dynamic translator, which performs a runtime conversion. However, QEMU's dynamic translator cannot still convert all instructions(e.g., **syscall, div**). In order to execute these instructions, QEMU implements a helper library. Therefore, in our current implementation, our rewritten instruction still needs these helper libraries to produce execution effects. We do not attempt to substitute this implementation, and we believe that the size of the helper library can be optimized in future development.

Another reason for the increase in binary size is that our rewrite transformation is based on instruction semantics. In our implementation, our IR lifter is based on execution semantics of instruction, which will bring the size increase when we transform these IRs to executable. So, we will update the IR optimization technique to reduce the code sizes in future work.

We also evaluate the cost of GRIN by comparing it with MULTIVERSE, and which mainly manifests in the size expansion of rewritten binaries. As shown in Figure 7, we compare the size expansion of rewritten binary of GRIN and MULTIVERSE. We find that the size expansion rewritten by GRIN has significantly reduced than MULTIVERSE. We also notice that some of the benchmarks have a higher size expansion than MULTIVERSE. That is because we rewrite some library functions as part of the rewritten binaries, but our average size expansion is lower than MULTIVERSE. We have analyzed the reason for expansion rewritten by GRIN in the previous section. In the rewrite implementation mechanism of MULTIVERSE, it rewrites the global mapping of libraries into rewritten binaries. If an original binary includes many libraries, the global mapping will dominate more space of rewritten binary. However, GRIN does not need to consider such issues.

7 CONCLUSION

We have presented GRIN, a novel binary rewriting tool that can precisely identify the memory address of each legal instruction in binary. In GRIN, we develop a dynamic execution technique and branch backtracking technique to address the challenges of identifying and rewriting binary. We rewrite instructions that we have executed into the rewritten binaries and do not consider relocation issues, guaranteeing correctness. As such, our solution can be applied to all security applications, such as binary hardening, instrumentation, and security policy reinforcement. Our experiments show that GRIN can successfully rewrite all the binaries we have tested (the whole set of GNU core utilities, SPECint 2006 benchmark, and realworld binaries) and correctly identify all legal instructions without redundancy. Therefore, GRIN can potentially be used as foundation work for binary-based software applications.

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China (No. 62172407), and the Youth Innovation Promotion Association CAS.

REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” ACM Transactions on Information and System Security (TISSEC), vol. 13, no. 1, pp. 1–40, 2009.

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

[3] D. Andriesse, X. Chen, V . V an Der V een, 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), 2016, pp. 583–600.

[4] P . Arafa, G. M. Tchamgoue, H. Kashif, and S. Fischmeister, “Qdime: Qos-aware dynamic binary instrumentation,” in 2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2017, pp. 132– 142.

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

[6] F. Bellard, “Qemu, a fast and portable dynamic translator.” in USENIX Annual Technical Conference, FREENIX Track, vol. 41, 2005, p. 46.

[7] A. B. Bergh, K. Keilman, D. J. Magenheimer, and J. A. Miller, “Hp-3000 emulation on hp precision architecture computers,” Hewlett-Packard Journal, vol. 38, no. 11, pp. 87–89, 1987.

[8] D. L. Bruening, Efficient, Transparent and Comprehensive Runtime Code Manipulation. Massachusetts Institute of Technology, 2004.

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

[10] B. H. Cogswell and Z. Segall, “Timing insensitive binary-to-binary migration across multiprocessor architectures,” in Proceedings of Third Workshop on Parallel and Distributed Real-Time Systems. IEEE, 1995, pp. 193–194.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P . Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” in USENIX security symposium, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[12] L. Davi, A.-R. Sadeghi, and M. Winandy, “Ropdefender: A detection tool to defend against return-oriented programming attacks,” in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 40–51.

[13] J. W. Davison, J. C. Knight, M. Co, J. D. Hiser, and A. Nguyen-Tuong, “Kevlar: Transitioning helix for research to practice,” UNIVERSITY OF VIRGINIA Charlottesville United States, Tech. Rep., 2016.

[14] Z. Deng, X. Zhang, and D. Xu, “Bistro: Binary component extraction and embedding for software security applications,” in European Symposium on Research in Computer Security. Springer, 2013, pp. 200–218.

[15] D. Developers, “Dyninst—dynamic instrumentation framework.” [Online]. Available: http://www.dyninst.org/parse,2020

[16] A. Di Federico, M. Payer, and G. Agosta, “rev. ng: a unified binary analysis framework to recover cfgs and function boundaries,” in Proceedings of the 26th International Conference on Compiler Construction, 2017, pp. 131–141.

[17] S. Dinesh, “Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization,” Ph.D. dissertation, Purdue University Graduate School, 2019.

[18] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, “Zipr: Efficient static binary rewriting for security,” in 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2017, pp. 559–566.

[19] J. K. Hollingsworth, B. P . Miller, and J. Cargille, “Dynamic program instrumentation for scalable performance tools,” in Proceedings of IEEE Scalable High Performance Computing Conference. IEEE, 1994, pp. 841–850.

[20] H. Inc, “Honeywell series 200 operating systems,” 1966, http://s3data.computerhistory.org/brochures/honeywell.osorientationmgmt.1966.102646090.pdf.

[21] C. Lattner and V . Adve, “The llvm compiler framework and infrastructure tutorial,” in International Workshop on Languages and Compilers for Parallel Computing. Springer, 2004, pp. 15–16.

[22] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, “Pebil: Efficient static binary instrumentation for linux,” in 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). IEEE, 2010, pp. 175–183.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V . J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” Acm sigplan notices, vol. 40, no. 6, pp. 190–200, 2005.

[24] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh, “Bird: Binary interpretation using runtime disassembly,” in International Symposium on Code Generation and Optimization (CGO'06). IEEE, 2006, pp. 12–pp.

[25] N. Nethercote and J. Seward, “V algrind: A framework for heavyweight dynamic binary instrumentation,” in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. New Y ork, NY , USA: Association for Computing Machinery, 2007, p. 89–100.

[26] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, “Cfi care: Hardwaresupported call and return enforcement for commercial microcontrollers,” in International Symposium on Research in Attacks, Intrusions, and Defenses. Springer, 2017, pp. 259–284.

[27] P . O'Sullivan, K. Anand, A. Kotha, M. Smithson, and A. D. Keromytis, “Retrofitting security in cots software with binary rewriting,” in Ifip International Information Security Conference, 2011.

[28] M. Payer, A. Barresi, and T. R. Gross, “Fine-grained control-flow integrity through binary hardening,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9148, no. i, pp. 144–164, 2015.

[29] M. Payer and T. R. Gross, “Fine-grained user-space security through virtualization,” ACM SIGPLAN Notices, vol. 46, no. 7, pp. 157–168, 2011.

[30] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, “X-force: Forceexecuting binary programs for security applications,” Proceedings of the 23rd USENIX Security Symposium, pp. 829–844, 2014.

[31] R. Qiao and R. Sekar, “Function interface analysis: A principled approach for function recognition in cots binaries,” in 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2017, pp. 201–212.

[32] T. Romer, G. V oelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of win32/intel executables using etch," in Proceedings of the USENIX Windows NT Workshop, vol. 1997, 1997, pp. 1–8.
[33] K. Scott, N. Kumar, S. V elusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in International Symposium on Code Generation and Optimization, CGO 2003, 2003.
[34] Y . Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, pp. 138–157.
[35] R. L. Sites, A. Chernoff, M. B. Kirk, M. P . Marks, and S. G. Robinson, "Binary translation," Communications of the ACM, vol. 36, no. 2, pp. 69–81, 1993.
[36] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," in Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, 1994, pp. 196–205.
[37] P . Team, "Pax address space layout randomization (aslr)," 2003, https://pax.grsecurity.net/docs/aslr.txt.
[38] R. Wang, S. Yan, A. Bianchi, A. Machiry, and G. Vigna, "Ramblr: Making reassembly great again," in NDSS, 2017.
[39] S. Wang, P . Wang, and D. Wu, "Reassembleable disassembling," in 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 627–642.
[40] R. Wartell, V . Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," Proceedings of the ACM Conference on Computer and Communications Security, pp. 157–168, 2012.
[41] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," ACM Computing Surveys (CSUR), vol. 52, no. 3, pp. 1–37, 2019.
[42] L. Xu, F. Sun, and Z. Su, "Constructing precise control flow graphs from binaries," University of California, Davis, Tech. Rep, 2009.
[43] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in NDSS, 2015.
[44] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in 2013 IEEE Symposium on Security and Privacy. IEEE, 2013, pp. 559–573.
[45] M. Zhang, M. Polychronakis, and R. Sekar, "Protecting cots binaries from disclosure-guided code reuse attacks," in Proceedings of the 33rd Annual Computer Security Applications Conference, 2017, pp. 128–140.
[46] M. Zhang and R. Sekar, "Control flow integrity for {COTS} binaries," in Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), 2013, pp. 337–352.