Static Binary Rewriting Techniques
CS6999 Report Computer Science & Engineering Department
IIT MADRAS
May 2022

CS20D408
Sai Venkata Krishnan V

Contents

1 Introduction

Binary Rewriting is the art of modifying an existing binary file, in the absence of any related source code, all while still preserving its executable state and not breaking it. Binary analysis is analyzing the properties of the binary, example: deriving the Control Flow Graph(CFG), memory layout etc... This forms a key step in binary rewriting. Unlike source code, Binaries are the final artifact that gets executed on a system, and represents the *ground truth* of how the program would actually behave [1]. Programmer expectations of the source code semantics is usually not preserved during the compilation and linking stage, due to various optimizations enabled by the compiler which changes the final semantics. For example, the registers which are clobbered across function calls, the stack layout of arrays etc... All these attributes of a binary vary with the compiler used and optimizations enabled, even though it is compiled from the same source code. Binary rewriting/analysis has many use-cases, especially when the source code is not available:

1. Optimizing an existing binary: Examples include removing dead code, converting indirect control flow instructions to direct control flow instructions.
2. Instrumenting the binary for profiling and performance monitoring of the binary.
3. Reverse engineering proprietary binaries and patch them. This is especially useful when they are no longer supported, i.e. legacy code base.
4. Porting binaries from one Instruction Set Architecture(ISA) to another.
5. Hardening the binary: Adding security features like Control flow integrity(cite), Stack bound checks

1.1 Types of Binary Rewriting

Binary rewriting can be performed either statically or dynamically:

1. Static Binary Rewriting: Involves analyzing and instrumenting the binary statically.
2. Dynamic Binary Rewriting: Involves using runtime library or emulator to analyze and modify the original binary at runtime.

Static Binary Rewriting	**Dynamic Binary Rewriting**
Low runtime overhead	High runtime overhead
Requires CFG reconstruction	Does not require CFG
Original binary is modified	Instrumentation is transparent
No support for dynamic code generation	Supports dynamic code generation
Depends on sound disassembler for code extraction	Does not require any static disassembly

Table 1: Static vs Dynamic Binary Rewriting[2]

Table 1 lists the high level differences between Static and Dynamic binary rewriting techniques. Both the techniques are widely researched. Dynamic Instrumentation, though easy to implement and various use-cases, suffers from very high overheads which hinders there adoption in real-world scenarios. Static instrumentation comparatively performs better and is more adoptable.

2 Static Binary Rewriting Steps

Any static binary rewriting technique involves the following four steps:

1. **Disassembly**: This is the first and crucial step which involves extracting the code section from the binary file for further analysis. It is important for the disassembler to extract all code bytes from the binary inorder for sound static binary analysis.

2. **Binary Analysis**: Static analysis on the binary is performed to build the control flow graph, memory layout of stack and heap, type recovery of functions etc... The extent to which the binary analysis is to be performed and how accurate it has to be depends on the specific use-case and the granularity at which binary instrumentation is to be supported.

3. **Binary Instrumentation**: The results from binary analysis can be used to modify the binary, which includes adding any new instruction or rearrange existing code layout of the binary. Binary analysis is crucial in preventing the binary from breaking when moving instructions around. Especially data references, indirect control flow targets, PC-relative addresses have to be adjusted accordingly to make sure binary executes properly.

In the rest of the report we will discuss challenges in each step and different static techniques developed. We will concentrate only on benign binaries, stripped and non-stripped. Obfuscated binaries are no taken into account in this report.

3 Disassembly

Disassembly of the binary involves extracting all the code section bytes present in the binary. The main challenge in static disassembly is to figure which are the code bytes and which are the data(non-code) bytes in a binary file. The common assumption is that code section would be contiguously placed separately from the data section. But this assumption is not preserved in some compilers. For example, Jump-tables(used for switch statement) are inlined within the code section. Given, a 32-bit or 64-bit value in a binary file, it is very difficult to differentiate whether the value represents a code or data. Moreover this problem becomes more acute in CISC architectures like x86 which are dense and where every byte can be a start of a valid instruction. Static Disassembly techniques include the following:

3.1 Linear Disassembly

This is the naive way of disassembly, where disassembly starts from the code section entry point, and linearly all the instructions are disassembled. GNU's objdump follows this method. This method fails in the presence of inline data like jump-tables embedded in the code section. For stripped binaries or where the code section starting address is unknown, the disassembly starts from binary file beginning. (diagram)

3.2 Recursive Disassembly

This technique was developed to skip the inlined data embedded in the code section. Disassembly starts from the known entry points like *_start*, *main* functions and continues with linear disassembly

until a control flow instruction is reached. Instead of the continuing disassembling after the control flow instruction, the recursive disassembler follows the control-flow and starts disassembling the targets. By following this method, any inline data which is present is skipped and not included as part of the code section. But two main difficulties arise:

1. The recursive disassembler would fail when it reaches indirect control flow instructions. Some heuristics need to be used to identify the indirect control flow targets and continue disassembly from there [3]. This leads to less code coverage when compared to the Linear disassembler.

2. Function code which is not reachable by any direct control flow instructions, but reachable indirectly, can be missed out in the final extracted code section resulting in broken binary for analysis.

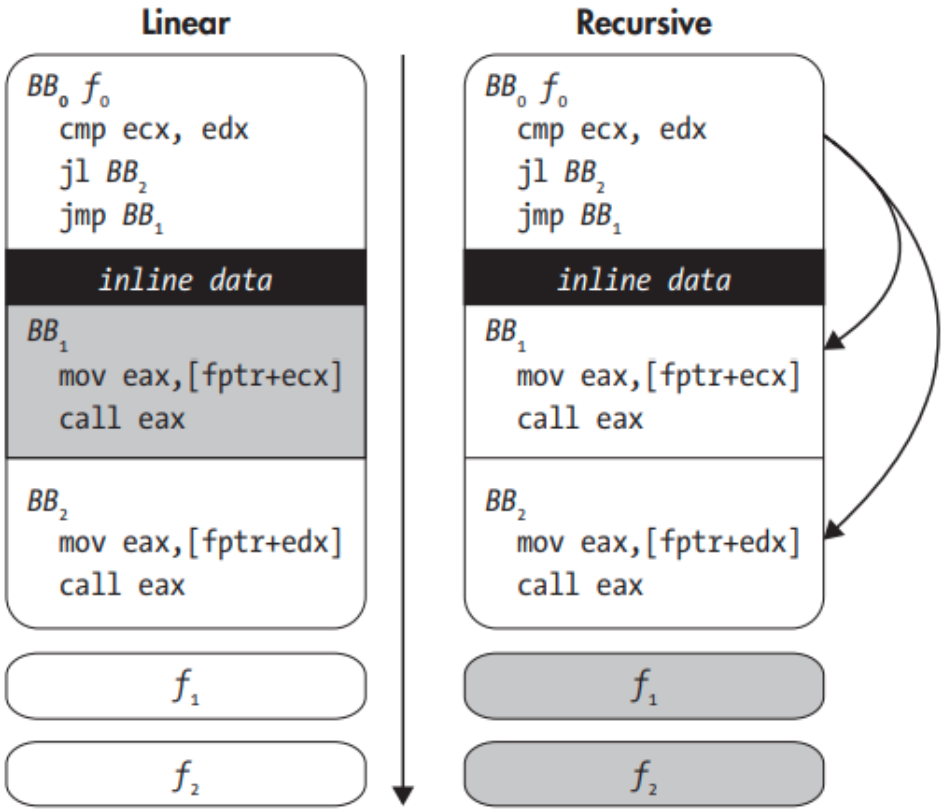

Figure 1: Linear & Recursive Disassembly [2]

3.3 Hybrid Disassembly

Static Hybrid Disassembler [3] tries to utilize both Linear as well as Recursive Disassembler and switch between them whenever the either one fails. It uses some heuristics to collect the list of potential indirect control flow targets. Heuristics include:

1. Identifying code pointers in the binary file, which would usually be loaded into registers for indirect control flow instructions. Any constant present in the file within the code section bounds would be treated a code pointer.

2. Identifying possible jump tables and deriving the jump destination targets

The first heuristic will fail in the presence of floating point constants [4] where the floating point constants are stored in binary file in a specific byte format which when read would be misinterpreted as a code pointer.

3.4 Superset Disassembly

This technique [5] tries to over-approximate the available code-section by performing an iterative linear disassembly from each offset of the binary file starting from 0. This results in disassembling the inlined-data in the code section also, but does not leave out any code bytes. This technique was mainly developed to handle dense CISC architectures and inlined-data. The rewriting technique based on this specific disassembly technique discussed in 5.2. The Datalog Disassembler [6] also uses this technique, where it parallelizes the disassembly implementation by utilizing the Datalog language and specifying the disassembly algorithm as set of rules and facts.

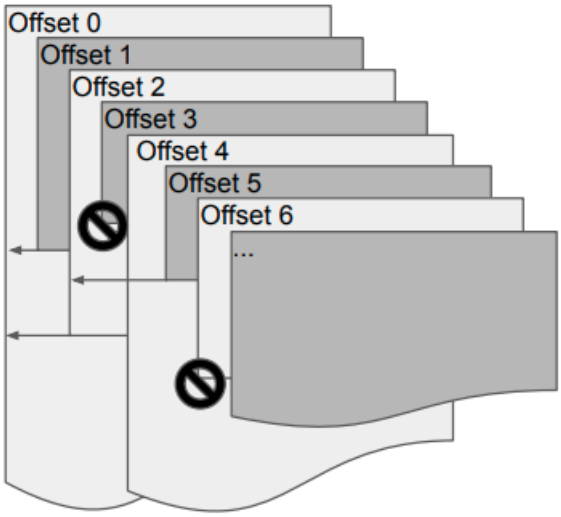

Figure 2: Superset Disassembly [5]

4 Binary Analysis

Binary analysis works upon the disassembled binary file to retrieve various properties of the binary, not limited to:control flow graph, function boundaries identification, basic block identification, stack layout, probable heap layout, type recovery of functions and callsites, possible uninitialized variables present in the stack/heap, alias analysis etc... The extent to which Binary analysis is performed and how accurate it should be depends on the granularity of the instrumentation that has to be supported. At a very coarse level, if a new code region that has to be executed at the

start of the program has to be inserted, then no complex analysis needs to be performed. Just a new code section can be appended to the existing binary file and added in the list of *.init* section which denotes the list of functions that has to be executed at the startup. Whereas, if you want to add some code at the start of each function, then is important to identify function boundaries. If individual instructions need to be modified, then the control flow graph and data layout of the binary should be analyzed, since moving individual instructions would break data references which are PC-relative and runtime-computed indirect control flow targets. The common techniques used in Binary Analysis includes:

1. **Value Set Analysis**: Value Set Analysis [7] tries to generate a map associating each memory location with a set of possible values. These values denote an abstract memory model and represent memory regions like stack, heap, global data section. The downside with Value Set Analysis is that, precise value sets are difficult to obtain and the time taken to obtain value sets are high. To optimize the Value Set Analysis, Ramblr [4] for instance uses a localized variant of Value Set Analysis, where the analysis is run only on the binary slice obtained w.r.t. specific memory location(which includes only the instructions which affect that memory cell or register) and not on the entire binary at an instance. This reduces the iterations taken for the Value Set Analysis to converge.

2. **Symbolization**: Symbolization is a key technique used widely, where constants in the binary are segregated into whether they are code or data pointers or literal constants. The code/-data pointer constants are symbolized so that when the binary is modified, these dependent constants can also be easily referenced and modified.

3. There are many other analysis techniques used: Call graph analysis, data dependency analysis, jump table extraction and Vtable extraction(in C++ binaries).

Analyzing a binary has many challenges, especially when compiler optimizations are enabled. A few are listed discussed below [[8], [9]]:

1. Function boundary identification is tricky in optimized binaries, where the function prologue and epilogue are optimized away in the binary. Function boundary detection based on prologue/epilogue detection would fail in such scenarios.

2. Overlapping functions are where when two functions share some code bytes among them. For example: Function 2 can be part of Function 1, where both functions share the return epilogue. In these cases it is difficult to extract function boundaries.

3. Jump table and Vtable extraction is important for symbolizing the binary. When optimizations are enabled, these static tables are interleaved within the code section, thereby making it difficult to identify them.

4. Statically linked code and Position Independent Code(PIC) are used in real-world systems. Binary analysis schemes which explicitly rely on symbols exported due to PIC enabled, would fail when applied on statically linked code. Most binary rewriters are fine-tuned to work with PIC binaries, based on the fact that dynamically linked and shared libraries are more widely deployed due to their performance benefits and also security benefit by enabling ASLR by default.

5. PC relative data and code references which are computed at runtime should be handled so that memory addresses computed at runtime do not point to garbage memory cell. For instance, the Uroboros binary rewriter [10] assumes that data section need not be relocated and hence does not try to identify data references/pointers present in the binary file. But this fails in real-world scenarios where any metadata required at runtime by the modified binary has to be inserted in the *.data*/*.rodata* sections. For example, to support control flow integrity, metadata which lists valid targets associated with each control flow instruction should be inserted in the binary for runtime checking.

5 Binary Instrumentation

This is the final step in Static Binary Rewriting, where the binary is modified correctly with help of results obtained from binary analysis so as to generate a valid binary. There are different static binary instrumentation techniques, and each of them have their own pros and cons and rely on different binary analysis results. We will be discussing the common static binary instrumentation techniques:

5.1 Instruction Patching

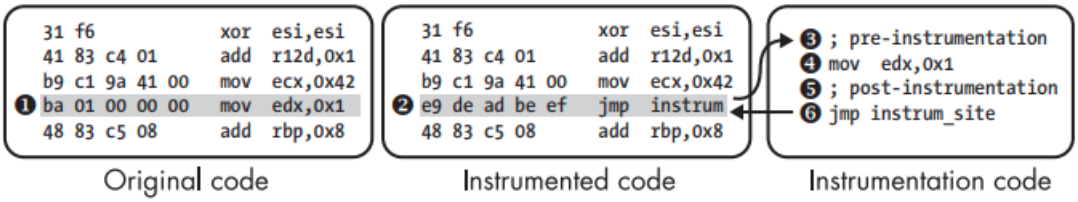

Figure 3: Naive Instruction Patching [2]

Instruction patching is a common binary rewriting technique which has less dependency on the Control Flow graph of the binary. The idea behind this approach, as shown in Figure 3, is that the instruction that needs to be monitored or modified is overwritten directly by a unconditional control-flow instruction which re-routes the control flow to a code stub(instrumentation stub) which houses the modified code byte which needs t be executed in its place at runtime. The code stub has to context save the registers based on its size, and should return back to the instruction following the overwritten instruction. The naive approach is to replace the instruction that needs to be modified by a jump instruction, where the location of the instrumentation stub is also hardcoded on the code byte. But this naive approach would fail when we have to say monitor/modify instructions which occupy less space than these jump instructions. For example, if we want to implement a shadow stack, then we have to monitor all return instruction, whose size is less than the jump instruction in x86 ISA. These problems are evident in CISC architecture which have varying instruction sizes. One way to handle this case in x86 is to use the *int3* interrupt instruction which raises a signal trap and the instruction itself is 1 byte in size and thus can be used in all scenarios. But this suffers from very high overhead, due to frequent context switching between user-space and kernel-space inorder to catch the signal and execute the instrumentation/monitoring code.

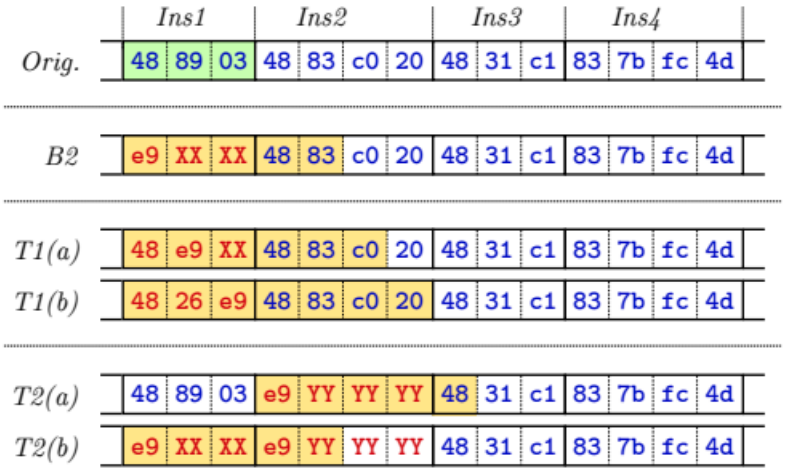

Figure 4: E9Patch Rewriter [11]

Instruction Punning [12] is patching technique for modifying multi-byte instructions, where the adjacent instruction bytes are utilized to form a overlapping jump instruction. For example in Figure 4 B2 represents a punned instruction, where the relative offset field in the overwritten instruction is constrained by the first two bytes of the adjacent instruction. This works in most cases, owing to the fact that process's virtual address space is sparsely populated. But this fails in cases where the relative offset points to an already allocated memory space. To circumvent this, e9patch rewriter [11] has developed some tactics to aid instruction punning. The first tactic, Padded Jumps makes use of redundant instruction prefixes, which can be padded along with punned jump instruction to obtain a constrained valid memory offset for the overwritten jump instruction, as shown in Figure 4 T1(a) and T1(b). Multiple redundant prefixes can also be used based on the size of the instruction that is being overwritten. To efficiently rewrite single-byte instructions, another tactic called Victim Eviction is used, where an adjacent multi-byte instruction is replaced by the Padded jump instruction, whose relative offset points to a instrumentation code which executes the victim instruction at runtime and returns back. By doing this, the actual instruction i.e. *Insn1* as shown in figure(ref:figure) obtains more wriggle space for finding a suitable relative offset for punning. The Victim Eviction technique results in executing two instrumentation stubs but can efficiently overwrite 1-byte instruction also and has less overhead when compared to the signal-based approach.

5.2 Trampoline based rewriting

Trampoline based rewriting technique, 5 is another naive technique, where instead of overwriting the instructions that has to be monitored, the whole code region is duplicated to form a new code region with modified instructions. The old *.text* section is retained inorder to ensure correct re-routing of indirect jump/call instructions whose addresses are computed at runtime. All instructions except the re-routing instructions in the old text/code section should be rewritten to point to trap/illegal instructions inorder minimize the attack surface. PIC instructions which read PC value by calling thunks should be handled properly such that instead of the new code section's PC value being read, the old text section's PC value is used for address computation at runtime. The advantage

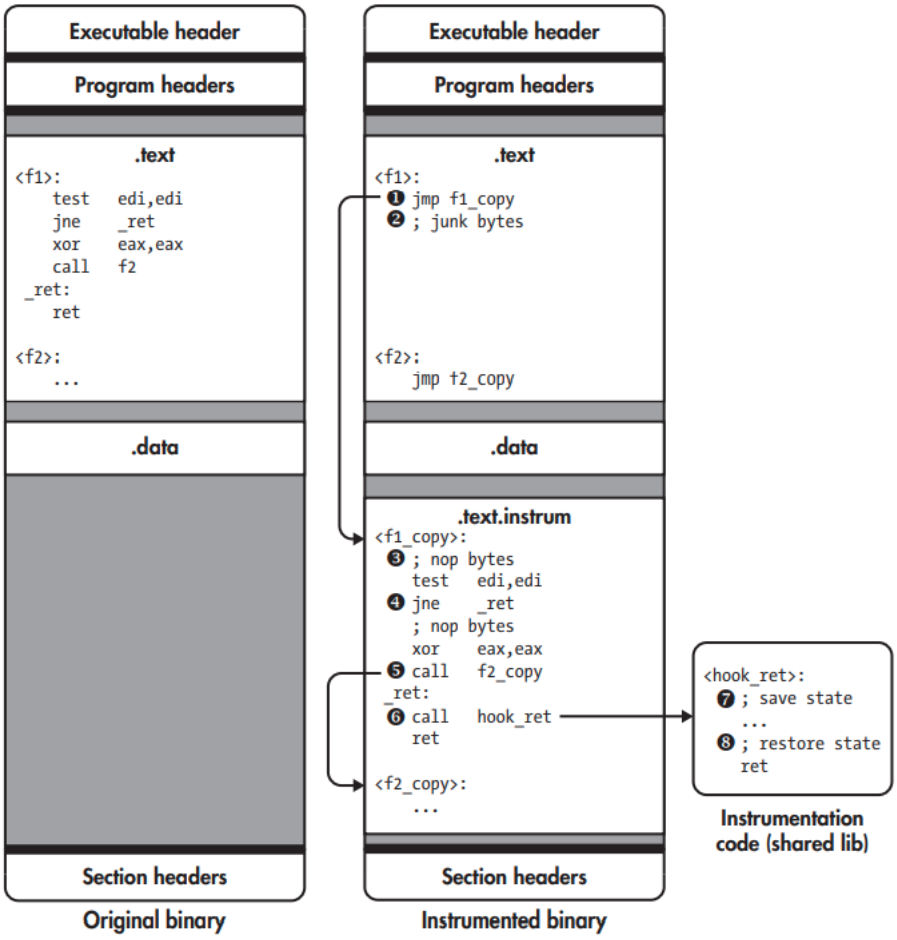

Figure 5: Trampoline Based Rewriter [2]

with Trampoline based approach is that, it is quite straight-forward and can be used to modify any instruction at any byte granularity. The disadvantage is that the switching between the new and old text sections pollutes the caches and memory and also the branch predictor which increases the memory and runtime overhead. Also, the binary file size is also doubled, as the old text section is also retained.

Multiverse Rewriter[5] is a trampoline/lookup based disassembler which uses the Superset disassembly to obtain the over-approximated set of instructions. Now for each library module which is to be rewritten, it maintains a local lookup table which maps the old code section address to the new code section address at byte granularity, as shown in Figure 6. The data section is retained as such for proper computations of indirect control flow targets at runtime. Direct call and jump instructions are overwritten directly, but indirect calls and jumps are transformed into a set of instructions which performs a local lookup of the target address to obtain the new relocated address of the target instruction that has to be executed. If the local lookup fails, then it might be a instruction present in another shared library. To support this, a global lookup table is populated at load time, which has all the local lookup table addresses of each library attached to the process. On a local lookup failure, the global lookup is utilized and checked whether the address falls within any

of the other local lookup table ranges, if yes, that local lookup table is searched and the relocated address is returned.

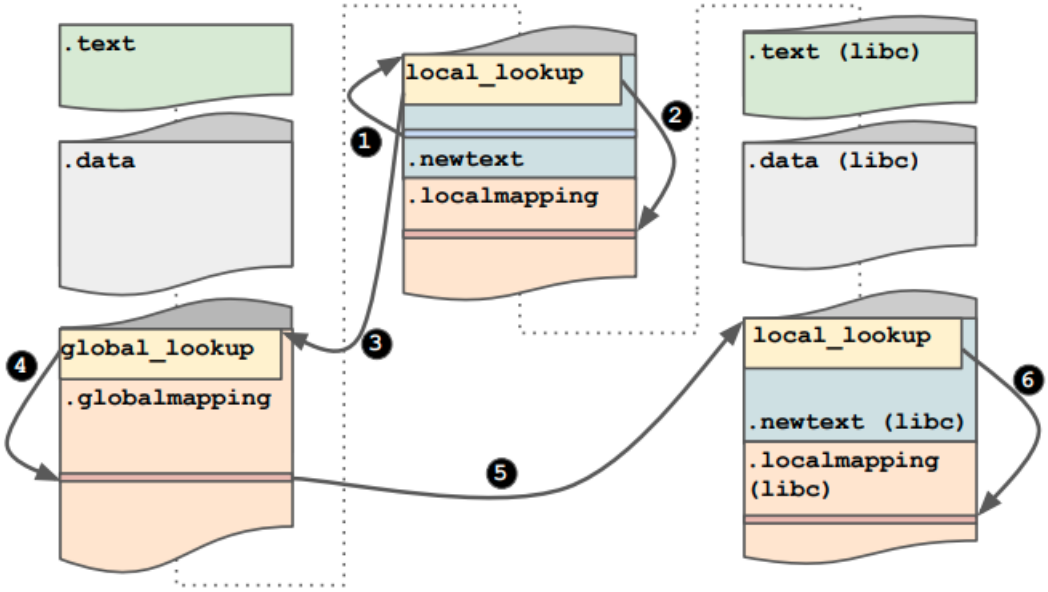

Figure 6: Multiverse Rewriter based on Superset Disassembly

5.3 Binary Lifting

Till now we have looked at binary instrumentation techniques which are ISA dependent. For example, in the Instruction Punning strategy, the various tactics like Padded Jumps and Victim Instruction Eviction are all dependent on specific x86 ISA. Moreover, if some special/redundant prefixes are made obsolete in future versions of the x86 ISA then these specific tactics might be constrained or fail altogether. Moreover for different ISAs, especially in embedded space, different techniques need to be developed explicitly for each ISA backend, which requires enormous engineering effort.

Binary Lifting based rewriter technique was developed to address the multiple-ISA backend issue. The general idea, is to lift the binary into an intermediate IR, with the machine instruction semantics captured and mapped to the IR instructions. Then various analysis passes, like data dependence analysis and alias analysis can be applied on the IR. The IR can be modified and assembled back into a binary. The general advantage with Binary Lifting approach is that, only the Lifter frontend has to be engineered to support different ISAs. The analysis passes are ISA agnostic and once designed would work for the IR irrespective of the ISA from which the IR was extracted. A key decision that governs the complexity of the Binary Lifter is the target IR to which the binary should be lifted. We discuss a few alternatives:

1. **High Level Languages i.e. Source language:** Decompiler does the reverse of a compiler, where given a binary, the source code is obtained. This is notoriously difficult and error prone. Common tools that support decompilation includes Ghidra [13] and IDAPro [14]. The

advantage with decompiled binary, is that it follows the source code's pseudocode notations and is easier for the reverse engineer to understand the code.

2. **High-level Intermediate IR**: Rather than lifting upto source code, it is easier to lift the binary upto an intermediate IR such as LLVM IR [15] or VEX IR(used by Valgrind) [16]. The advantage with this approach is that, these IR languages have a vast set of compiler analysis passes which can be directly reused to analysed the lifted binary. Complexities arise based on how much information has to be captured into the IR for meaningful analysis. For example, lifting to the LLVM IR from each target ISA is complex, and LLVM IR has such requires type annotation etc... , to be filled in for meaningful analysis. In CISC architecture like x86, the side-effects of an instruction also has to be mapped to the IR. Common examples of high-level IR binary lifters include RevNG [15] which targets the LLVM IR and Angr [16] targeting the VEX IR.

3. **Low level IR**: Lifting the binary to a low-level IR like LLVM's MachineInst is easier when compared to lifting it to higher-level LLVM IR. Custom IR's are developed to support efficient binary analysis. For example: Egalito [17] and Datalog Disassembler [17] lift the binaries to a custom IR, which is less expressive then high-level LLVM IR, but captures enough information in ISA agnostic way for suitable binary analysis. Information captured by these custom IRs include, the identified basic blocks, control-flow between them, functions identified, any code/data symbols found in the binary. All these information would be utilized to perform IR analysis and modify the IR and convert it back to an executable binary. RevNG rewriter [15] doesn't lift the binary directly to LLVM IR. Instead it uses an intermediate low-level IR called TCG ISA used by QEMU emulator, which supports varying target architectures. Lifting from this low-level IR to LLVM IR is relatively easier.

4. **Reassembleable Disassembly**: Recently, some binary lifters like Ramblr [4], Uroboros [10], Retrowrite [18] and Datalog Disassembler [6] aim to lift the binary to the target architecture(ISA's) assembly language itself and not any custom or low-level IR. The advantage is that the assembly language is weakly-typed and is a easier target for binary lifting. The key to achieve reassembleable disassembly is to ensure that the assembly code which is obtained on binary lifting is relocatable. To achieve this, all code and data pointer present in both data and code regions of the executable have to be symbolized, so that while reassembling the lifted assembly code, the linker relocation techniques can be applied. Uroboros [10] was the first one to propose this technique. Datalog Disassembler [6] is the current state-of-the-art reassembleable disassembler, supporting both PIC and non-PIC stripped/non-stripped binaries. Instead of relying on Value Set Analysis, which has high runtime overheads, Datalog Disassembler encodes the binary in Datalog as a set of predicates and rules, and utilizes various weighted heuristics to classify whether a literal constant points to a data region or code region or is a literal string/float/int constant itself.

The advantage with Binary Lifting technique is that, the code locality is not changed , in contrast to the instruction patching and trampoline approach. Also various link time and compiler optimizations can be enabled to make the modified-binary perform efficiently.

6 Conclusion

We have seen the various phases involved in successfully rewriting a binary, the various techniques used in each phase and challenges faced in each step. Each analysis and instrumentation phase has its own advantages and disadvantages and is well-suited for different use-case scenarios.

References

[1] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, aug 2010. [Online]. Available: https://doi.org/10.1145/1749608.1749612

[2] D. Andriesse, *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly.* No Starch Press, 2018.

[3] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium (USENIX Security 13).* Washington, D.C.: USENIX Association, Aug. 2013, pp. 337–352. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

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

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

[6] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, Aug. 2020, pp. 1075–1092. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[7] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *CC*, 2004.

[8] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 24–35. [Online]. Available: https://doi.org/10.1145/2931037.2931047

[9] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *ACM Comput. Surv.*, vol. 52, no. 3, jun 2019. [Online]. Available: https://doi.org/10.1145/3316415

[10] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, p. 627–642.

[11] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 151–163. [Online]. Available: https://doi.org/10.1145/3385412.3385972

[12] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton, "Instruction punning: Lightweight instrumentation for x86-64," *SIGPLAN Not.*, vol. 52, no. 6, p. 320–332, jun 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062344

[13] NSA. Ghidra 2019. [Online]. Available: https://www.nsa.gov/resources/everyone/ghidra/

[14] Hex-rays. Hex-rays: The ida pro disassembler and debugger. [Online]. Available: https://www.hex-rays.com/products/ida

[15] 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*, ser. CC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 131–141. [Online]. Available: https://doi.org/10.1145/3033019.3033028

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

[17] 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*, ser. CC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 131–141. [Online]. Available: https://doi.org/10.1145/3033019.3033028

[18] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," 05 2020, pp. 1497–1511.